OSP-9961 US

日 本 国 特 許 庁
PATENT OFFICE
JAPANESE GOVERNMENT

別紙添付の書類に記載されている事項は下記の出願書類に記載されている事項と同一であることを証明する。
This is to certify that the annexed is a true copy of the following application as filed with this Office.

出願年月日
Date of Application: １９９９年１２月 ２日

出願番号
Application Number: 平成１１年特許願第３４３９５９号

出願人
Applicant (s): 日本電気株式会社

２０００年 ９月 ８日

特許庁長官
Commissioner,
Patent Office 及川耕造

出証番号 出証特２０００－３０７１２５０

【書類名】　特許願

【整理番号】　74510187

【提出日】　平成11年12月 2日

【あて先】　特許庁長官　殿

【国際特許分類】　G06F 17/00

【発明の名称】　オブジェクト指向言語によるシステムのシミュレーション方法、装置及びそのプログラムを記録した記録媒体

【請求項の数】　10

【発明者】

【住所又は居所】　東京都港区芝五丁目７番１号　日本電気株式会社内

【氏名】　笠　浩史

【発明者】

【住所又は居所】　東京都港区芝五丁目７番１号　日本電気株式会社内

【氏名】　加藤　雄祐

【特許出願人】

【識別番号】　000004237

【氏名又は名称】　日本電気株式会社

【代理人】

【識別番号】　100108578

【弁理士】

【氏名又は名称】　高橋　詔男

【代理人】

【識別番号】　100064908

【弁理士】

【氏名又は名称】　志賀　正武

【選任した代理人】

【識別番号】　100101465

【弁理士】

【氏名又は名称】　青山　正和

【選任した代理人】

【識別番号】　100108453

【弁理士】

【氏名又は名称】　村山　靖彦

【手数料の表示】

【予納台帳番号】　008707

【納付金額】　21,000円

【提出物件の目録】

【物件名】　明細書　1

【物件名】　図面　1

【物件名】　要約書　1

【包括委任状番号】　9709418

【プルーフの要否】　要

【書類名】　明細書

【発明の名称】　オブジェクト指向言語によるシステムのシミュレーション方法、装置及びそのプログラムを記録した記録媒体

【特許請求の範囲】

【請求項１】　複数の回路モジュールからなるシステムをソフトウェアによってシミュレーションするシミュレーション方法において、

オブジェクト指向言語を用い、

基本となる回路モジュールをクラスとして記述した複数の回路基底クラスをあらかじめライブラリとして用意しておき、

ライブラリとして用意されている回路基底クラスを継承することで、シミュレーションする回路モジュールをクラスとして記述し、

クラスとして記述した複数の回路モジュールを組み合わせることで、シミュレーションするシステムの記述を行う

ことを特徴とするシミュレーション方法。

【請求項２】　ライブラリに用意されている回路基底クラスを継承することで、システムをクラスとして記述する

ことを特徴とする請求項１記載のシミュレーション方法。

【請求項３】　少なくとも、クロック信号に非同期で動作する回路を含む回路の性質を有するコンポーネントクラスと、コンポーネントクラスを基本クラスとする派生クラスであって、クロックに同期して動作する回路の性質を有する同期モジュールクラスとを、回路基底クラスとして、あらかじめライブラリに用意しておく

ことを特徴とする請求項１又は２記載のシミュレーション方法。

【請求項４】　さらに、バスの性質を有するバスクラスと、バスマスタの性質を有するバスマスタクラスと、バススレーブの性質を有するバススレーブクラスとを、同期モジュールクラスの派生クラスとして記述された回路基底クラスとして、あらかじめライブラリに用意しておく

ことを特徴とする請求項３記載のシミュレーション方法。

【請求項５】　さらに、同期モジュールクラスを基本クラスとするものであ

って、バスマスタインターフェースの性質を有するバスマスタインターフェースクラスと、バススレーブインターフェースの性質を有するバススレーブインターフェースクラスとを、回路基底クラスとして、あらかじめライブラリに用意しておく

ことを特徴とする請求項４記載のシミュレーション方法。

【請求項６】　さらに、同期モジュールクラスを基本クラスとするものであって、中央処理装置の性質を有する中央処理装置クラスを、回路基底クラスとして、あらかじめライブラリに用意しておく

ことを特徴とする請求項３記載のシミュレーション方法。

【請求項７】　さらに、同期モジュールクラスを基本クラスとするものであって、バスを含む回路の１階層の性質を有する階層クラスを、回路基底クラスとして、あらかじめライブラリに用意しておく

ことを特徴とする請求項３～６のいずれか１項に記載のシミュレーション方法。

【請求項８】　さらに、バススレーブクラスを基本クラスとするものであってメモリの性質を有するメモリクラスを、回路基底クラスとして、あらかじめライブラリに用意しておく

ことを特徴とする請求項４又は５記載のシミュレーション方法。

【請求項９】　コンピュータを用い、請求項１～８のいずれか１項に記載のシミュレーション方法を実行するシミュレーション装置。

【請求項１０】　請求項１～８のいずれか１項に記載のシミュレーション方法をコンピュータを用いて実行する際に用いられるプログラムを記録した電子計算機読み取り可能な記録媒体。

【発明の詳細な説明】

【０００１】

【発明の属する技術分野】

本発明は、例えばＳＯＣ（System On a Chip）等の複数の回路モジュールから構成されている大規模なシステムをソフトウェアによってシミュレーションする際に用いて好適なオブジェクト指向言語を用いたシステムのシミュレーション方

法並びにシミュレーション装置及びシミュレーションプログラムを記録した記録媒体に関する。

【０００２】

【従来の技術】

システムを設計するときなどに用いられる従来のソフトウェア・シミュレータの一つにＨＤＬ（Hardware Description Language；ハードウェア記述言語）を用いたものがある。図３７は、従来のＨＤＬにおける回路モジュール間の接続例を示す説明図であり、（ａ）はＨＤＬによる回路モデルの記述例を、そして（ｂ）は（ａ）の記述例に対応する模式図を示している。ＨＤＬで記述された回路モジュールの入出力は、各端子によって行われる。回路モジュールの間の通信は、信号線を端子に接続することによって行われる。例えば、図３７（ｂ）に示すように、回路モジュールＣの中に回路モジュールＡと回路モジュールＢが存在する場合に、回路モジュールＡの端子ａ１と回路モジュールＢの端子ｂ１を接続するには、回路モジュールＣの中に信号線ｃ１を作り（図３７（ａ）の①）、回路モジュールＡの端子ａ１を回路モジュールＣの信号線ｃ１に接続し（②）、そして、回路モジュールＢの端子ｂ１を回路モジュールＣの信号線ｃ１に接続することで行われる（③）。信号線が接続されると、一方のシミュレーションモデルがその信号線をドライブし、他方のシミュレーションモデルがその信号値を観測することで、信号伝播を行うことができる。回路モジュール間を正しく通信させるには、一方の回路モジュールの端子を他方の回路モジュールの端子に正しく接続する必要がある。

【０００３】

ここで、複数の回路モジュールからなるシステムを設計する場合に、ＨＤＬ等によるソフトウェアシミュレータを用いるときの一般的な設計手順について説明する。まず、システムを複数の回路モジュールに分割する。このとき、後で結合することを考え、各回路モジュールのインターフェースの規約を定める。インターフェースの規約とは、ある回路モジュールのどの端子がどのような信号線幅を持ち、どのような役割をするかを記したものである。この規約は、複数の設計者の間で共有できるようにするため、自然言語で文書に記述されるか、口頭で伝え

られ、各回路モジュールはこの規約に従って作成される。

【０００４】

すべての回路モジュールが作成されたら、各回路モジュールが互いに通信を行うように組み合わせて、ＨＤＬによってシステムを作成（記述）する。各回路モジュールが規約に従っているかどうかは、目視か、実際にシステムのシミュレーションを行って結果を観測することで行われる。シミュレーション結果が正しいことが確認されたら、このＨＤＬ記述を元に実際の回路を作成する。

【０００５】

【発明が解決しようとする課題】

ところで、上記のような従来のＨＤＬ等によるソフトウェアシミュレーションは、もともと小規模な回路モジュールあるいは回路システムを検証するために開発されてきたものである。したがって、高い集積度を有するＬＳＩ（Large Scale Integration）等のＳＯＣに適用するのには、いくつかの課題が生じてきている。

【０００６】

例えば、２つの回路モジュールを接続する場合には、回路モジュール双方の端子の役割を確認する必要がある。しかしながら、従来のＨＤＬ等によるシミュレータでは、この役割を確認するための作業が自動的に行われるようにはなっていなかった。例えば、ＨＤＬでは端子の名前を自由に設定できるため、クロック、リセットなどの、ある決まった意味を持つ端子について、その名前が回路モジュール毎にばらばらになる可能性がある。その場合、端子間を信号線で接続する際には、回路モジュール毎にどの端子がどの意味を持つかを、ＨＤＬ記述と、他の文書などを、人間が目視により照合して確認する必要がある。このため、複数の回路モジュールを接続してシステムを組み上げるときに、モジュール間の接続確認のために設計者に面倒な手作業を強いることとなっていた。

【０００７】

また、回路モジュールを接続する場合には、各回路モジュールごとに接続先の回路モジュールのすべての端子について記述する必要があるため、一旦記述した接続を変更する場合等に多数箇所の記述を変更する必要があるという課題があっ

た。例えばＨＤＬでは、仮にある回路モジュールのクロック端子だけに入力を与えれば良い場合でも、その回路モジュールのすべての端子について、それらを接続するかしないかを記述する必要がある。つまり、接続先を変更する度に、接続先の回路の詳細な情報を記述しなくてはならない。このため、システムを構成する回路モジュールを入れ換える場合等に、面倒な作業が必要となっていた。

【０００８】

一方、上述したような従来のＨＤＬ等によるソフトウェアシミュレーションと異なる技術であって、複数のモジュールによるシステムの開発により適したシミュレーション方法を提供するための技術が、文献『OO-VHDL Object-Oriented Extensions to VHDL』（IEEE Computer, Oct. 1995, pp. 18-26）に提案されている。この文献に記載されているシミュレーション方法は、従来のＨＤＬにオブジェクト指向言語の特徴を組み入れた新たな言語を用い、各モジュールおよび各モジュール間の接続を記述するものである。例えば、“クラス”や“継承”といったオブジェクト指向言語の特徴を利用することで、各モジュールの記述や再利用をより容易にしようとする技術が記載されている。しかしながら、この文献には、それに記載されているシミュレーション方法あるいはモジュールの記述方法を種々の回路モジュールを複数接続した大規模なシステムに適用する場合の具体的な手法が示されていなかった。

【０００９】

また、大規模なシステムのシミュレーションに、オブジェクト指向言語の１つであるＣ＋＋言語を使用する方法が、文献『ＳＯＣの事前検証を実現するＣ＋＋シミュレータ「ClassMate」』（電子情報通信学会、信学技報VLD98-46, 1998-09）に提案されている。しかしながら、この文献に記載されているシミュレーション方法あるいはモジュールの記述方法は、オブジェクト指向言語の利点を十分に利用していなかった。そのため、各モジュールの再利用や各モジュール間の接続に関しては、ＨＤＬと同様の問題が生じていた。

【００１０】

本発明は、具体的に、従来の技術にくらべ、複数モジュールによって構成されるシステムをソフトウェアによってシミュレーションする際に、システムおよび

回路モジュールの記述ならびに各モジュール間の接続に関する記述や、システム、回路モジュールの拡張、変更等を、より容易にかつ確実に行うことができるようにするためのオブジェクト指向言語によるシステムのシミュレーション方法並びにシミュレーション装置及びシミュレーションプログラムを記録した記録媒体を提供することを目的とする。

【００１１】

【課題を解決するための手段】

上記課題を解決するため、請求項１記載の発明は、複数の回路モジュールからなるシステムをソフトウェアによってシミュレーションするシミュレーション方法において、オブジェクト指向言語を用い、基本となる回路モジュールをクラスとして記述した複数の回路基底クラスをあらかじめライブラリとして用意しておき、ライブラリとして用意されている回路基底クラスを継承することで、シミュレーションする回路モジュールをクラスとして記述し、クラスとして記述した複数の回路モジュールを組み合わせることで、シミュレーションするシステムの記述を行うことを特徴としている。

【００１２】

また、請求項２記載の発明は、ライブラリに用意されている回路基底クラスを継承することで、システムをクラスとして記述することを特徴としている。また、請求項３記載の発明は、少なくとも、クロック信号に非同期で動作する回路を含む回路の性質を有するコンポーネントクラスと、コンポーネントクラスを基本クラスとする派生クラスであって、クロックに同期して動作する回路の性質を有する同期モジュールクラスとを、回路基底クラスとして、あらかじめライブラリに用意しておくことを特徴としている。

【００１３】

また、請求項４記載の発明は、さらに、バスの性質を有するバスクラスと、バスマスタの性質を有するバスマスタクラスと、バススレーブの性質を有するバススレーブクラスとを、同期モジュールクラスの派生クラスとして記述された回路基底クラスとして、あらかじめライブラリに用意しておくことを特徴としている。また、請求項５記載の発明は、さらに、同期モジュールクラスを基本クラスと

するものであって、バスマスタインターフェースの性質を有するバスマスタインターフェースクラスと、バススレーブインターフェースの性質を有するバススレーブインターフェースクラスとを、回路基底クラスとして、あらかじめライブラリに用意しておくことを特徴としている。

【００１４】

また、請求項６記載の発明は、さらに、同期モジュールクラスを基本クラスとするものであってバスマスタの性質を有するバスマスタクラスと、バスマスタクラスを基本クラスとするものであって中央処理装置の性質を有する中央処理装置クラスとを、回路基底クラスとして、あらかじめライブラリに用意しておくことを特徴としている。また、請求項７記載の発明は、さらに、同期モジュールクラスを基本クラスとするものであって、バスを含む回路の１階層の性質を有する階層クラスを、回路基底クラスとして、あらかじめライブラリに用意しておくことを特徴としている。また、請求項８記載の発明は、さらに、バススレーブクラスを基本クラスとするものであってメモリの性質を有するメモリクラスを、回路基底クラスとして、あらかじめライブラリに用意しておくことを特徴としている。

【００１５】

また、請求項９記載の発明は、コンピュータを用い、請求項１～８のいずれか１項に記載のシミュレーション方法を実行するシミュレーション装置である。また、請求項１０記載の発明は、請求項１～８のいずれか１項に記載のシミュレーション方法をコンピュータを用いて実行する際に用いられるプログラムを記録した電子計算機読み取り可能な記録媒体である。

【００１６】

上述したように、本発明は、システムのシミュレーションモデルを記述する方法に主要な特徴を有している。本発明では、オブジェクト指向言語を用いてシステムを記述することを特徴としている。これにより、回路モジュールが持つべき性質を、クラス・ライブラリとして用意することが可能となる。また、ユーザ（設計者）が回路モジュールを記述する際は、この性質を継承して記述することを可能としている。これらにより、（１）回路モジュールの差し替えが容易になる、回路モジュール間の接続の正しさが自動で検査される、（２）情報表示の仕組

みを共通に使用することができる、（３）シミュレーション制御の仕組みを共通に使用することができる、（４）回路のインターフェース部の記述が十分であるかどうかを自動で検査できる等の利点を得ることが可能となる。

【００１７】

【発明の実施の形態】

以下、図面を参照して本発明によるシミュレーション方法の実施の形態について説明する。なお、本実施の形態においては、シミュレーションモデルをＣ＋＋言語を用いて記述するものとする。ここで、Ｃ＋＋言語およびこの説明において用いる用語の定義について説明する。

【００１８】

［Ｃ＋＋言語］

【００１９】

Ｃ＋＋は、プログラミング言語の１つで、オブジェクト指向言語に分類される。Ｃ＋＋で用いられる用語に、クラス、メンバ関数（メソッド）、オブジェクト、継承、基本クラス、派生クラス、オーバーライド（オーバーロード）、純粋仮想関数等がある。クラスとは、ある性質を表すものである。この性質はメンバ関数などによって記述される。メンバ関数とは、クラスに対する操作を規定したものである。メンバ関数はメソッドと呼ばれることがある。一般に、クラスClassXにメンバ関数Function1という操作が許されているとき、クラスClassXはメンバ関数Function1を持つという。オブジェクトとは、クラスを具現化した実体である。例えば、上記クラスClassXを具現化してオブジェクトObjectXを得たとき、オブジェクトObjectXをクラスClassXのオブジェクトという。オブジェクトはクラスに規定された性質を持つ。例えば、図１に示すＣ＋＋言語の記述例では、クラスClassAはメンバ関数Function1とFunction2を持つので（①）、ObjectAもFunction1とFunction2を持つ（②）。

【００２０】

継承とは、あるクラスの性質を別のクラスに引き継ぐことを言う。具体的には、あるクラスに定義されたメンバ関数がそのまま継承されたクラスにも定義される。例えば、上記クラスClassAを継承して別のクラスClassBを作成した場合（③

）、クラスClassAはメンバ関数Function1を持つと規定されているので、ClassBはFunction1について特に規定しなくても、暗黙的にFunction1を持つことになる（④）。

【００２１】

基本クラスとは、継承される側のクラスのことである。上記例では、クラスClassAはクラスClassBの基本クラスであるという（③）。派生クラスとは、継承する側のクラスのことである。上記例では、クラスClassBはクラスClassAの派生クラスであるという。派生クラスの定義は、③で示す文のように、定義する派生クラス名（クラスClassB）を予約語“class”にあとに記述し、それに続いて“：（コロン）public　基本クラス名（ClassA）”と記述することで行う。ここで“public”は派生方法（基本クラスのメンバーへのアクセス方法（公開派生））を指定する予約語である。派生クラスのオブジェクトは、基本クラスのオブジェクトとして扱うことができる。例えば上記の例において、クラスClassBのオブジェクトObjectBは、クラスClassAのオブジェクトとしても扱うことができる（⑤）。

【００２２】

オーバーライドとは、すでに基本クラスで定義されているメンバ関数の動作を、派生クラスで再定義することをいう。例えば図１において、クラスClassAではメンバ関数Function2がどのような動作するかが定義されているが（⑥の“…”の部分（ただし詳細な記述は省略））、派生クラスClassBではこの動作が好ましくないとする。このような場合、クラスClassBでメンバ関数Function2を再定義することで、メンバ関数Function2の動作を変更できる（⑦）。これを、クラスClassBでメンバ関数Function2をオーバーライドするという。クラスClassBでメンバ関数Function2をオーバーライドすれば、クラスClassBのメンバ関数Function2はクラスClassAのメンバ関数Function2と異なる動作をする。

【００２３】

あるクラスが持つメンバ関数のうち、そのクラスでは関数の処理の内容が定義されていないものを、純粋仮想関数と呼ぶ（⑧；“関数＝0”の形式で記述）。純粋仮想関数を持つクラスのオブジェクトを生成しようとするとコンパイル時に

エラーとなる（⑨；図１の“//”はＣ＋＋言語においてコメントを示す記号である。）。純粋仮想関数を持つクラスは継承とともに用いられる、派生クラス側で純粋仮想関数がオーバーライドされることを前提としている（（10））。

【００２４】

［前提事項］

【００２５】

次に、本実施の形態を実施する際、すなわちシミュレーションを行う際に前提となる事項について説明する。通常、システムを構築する場合には、役割毎にシステムをいくつかのブロックに分ける。これらのブロックを回路モジュールと呼ぶ。ここで、各回路モジュールが他の回路モジュールとどのような通信を行うかを決める。そして、これらの回路モジュールをそれぞれ個別に設計する。すべての回路モジュールの設計が終了した後、これらの回路モジュールを組み上げ、１つのシステムにする。システムはこれらの複数の回路モジュールが通信し合うことにより動作する。このようなシステムのシミュレーションを行う手順は、以下の通りである（図２）。

【００２６】

１．シミュレーション対象となるシステムをＣ＋＋言語により記述する。

【００２７】

（Ａ）各回路モジュールのクラスを宣言する（図２の①）。図２の例では、ユーザ作成クラスとして、ＣＰＵクラス１０１、バスマスタインターフェースクラス１０２、バスクラス１０３を作成している。

【００２８】

（Ｂ）システム全体のシミュレーションモデル（システム記述１０４）を作成する（②）。まず、システムに含まれる各回路クラスのオブジェクトを作成する（③）。次に、各回路オブジェクト間を接続し、回路オブジェクト間の通信を記述することで、システム全体のシミュレーションモデルを記述する（④）。回路オブジェクト間の接続は、一方の回路オブジェクトＡのポインタを他方の回路オブジェクトＢに渡すことにより行う。また、回路オブジェクト間の通信は、例えば、一方の回路オブジェクトＡの関数を他方の回路オブジェクトＢから実行するこ

とにより行う（図３参照）。図３は、オブジェクトObjectBがオブジェクトObjectAから引数ｉで示されるデータを取り込む（GetData）場合のＣ＋＋言語による記述例（ａ）とその模式図（ｂ）を示している。

【００２９】

（Ｃ）次に、回路の情報表示部を組み込む。シミュレーションによる事前検証の段階では、例えばＣＰＵやバスがどのような動作をしているかを知る必要がある。例えば、現在ＣＰＵはソフトウェアのどの部分を実行しているか、ＣＰＵ上で動いているソフトウェアの各変数はどのような値になっているか、バスの占有率はどれくらいか、バスをどのマスタが使用しているか、などである。これらの情報を表示する処理を行う記述もシステムの記述に付加する。図２では、あらかじめライブラリにとして用意しておいたＣＰＵの情報を表示するための記述を、ＣＰＵ情報表示部１０５として、また、バスの情報を表示するための記述を、バス情報表示部１０６として、それぞれ、必要箇所に、また必要な変更を加えて、システム記述１０４に取り込むこととしている。

【００３０】

２．シミュレーション制御部を組み込む。

【００３１】

シミュレーション実行中はユーザの指示に応じて、システムをリセットしたり動作させたりする必要がある。そこで、ユーザの指示に応じてシステムの制御を行う機構（シミュレーション制御部１０７）をシステムの記述１０４に付加することでシミュレータソース１０８を完成させる。

【００３２】

３．項目２．で記述したシミュレーションモデル（シミュレータソース１０８）をＣ＋＋コンパイラ１０９でコンパイルして、シミュレータの実行形式（シミュレータ１１０）を得る。

【００３３】

４．シミュレータ１１０を実行することにより、シミュレーションを行う。

【００３４】

シミュレーションによりシステムの動作が正しいことが確認されたら、このＣ

＋＋記述を元に実際の回路を作成する。

【００３５】

[本発明による実施の形態の具体的構成]

【００３６】

本実施の形態で用いるクラスについて説明する。本実施の形態では、図４のようなクラスを、クラス・ライブラリとして用意する。本実施の形態では、図４に示すように、回路全般に共通する性質を有するクラスCmComponent（コンポーネントクラス）、クロックに同期して動作する回路の性質を有するクラスCmSyncModule（同期モジュールクラス）、バスマスタのメイン部の性質を有するクラスCmBusMaster（バスマスタクラス）、バススレーブのメイン部の性質を有するクラスCmBusSlave（バススレーブクラス）、バスマスタインターフェースの性質を有するクラスCmBusMstIntf（バスマスタ・インターフェースクラス）、バススレーブインターフェースの性質を有するクラスCmBusSlvIntf（バススレーブ・インターフェースクラス）、バスの性質を有するクラスCmBusSystem（バスクラス）、ＣＰＵの性質を有するクラスCmCpu（ＣＰＵクラス）、メモリの性質を有するクラスCmMemory（メモリクラス）、そして、バスを含む回路の１階層の性質を有するクラスCmHier（階層クラス）を定義することとする。

【００３７】

本発明においては、あらかじめライブラリとして用意するこれらのクラスを、回路基底クラスと呼ぶ。これらのクラスの間には、図５に示す継承関係がある。すなわち、図５においては、クラスCmSyncModule（２０２）がクラスCmComponent（２０１）を継承し、それ以外の各クラス（クラスCmBusMaster（２０３）、クラスCmBusSlave（２０４）、クラスCmBusMstIntf（２０５）、クラスCmBusSlvIntf（２０６）、クラスCmBusSystem（２０７）、クラスCmHier（２０８））が、クラスCmSyncModule（２０２）を継承するように定義されている。なお、図５では、各クラスの継承関係とともに、各クラスが持つ主なメンバ関数の一覧を示しているが、メンバ関数の詳細については後述する。また、図５に示していないクラスCmCpu（クラス・ＣＰＵ）は、クラスCmBusMaster（２０３）を継承し、クラスCmMemory（クラス・メモリ）は、クラスCmBusSlave（２０４）を継承するよう

に定義されていることとする。

【００３８】

最初にシステムを回路モジュールに分割する段階では、各回路モジュールのシステムにおける役割はすでに決まっているはずである。そこで、システムの各回路モジュールに対応する回路クラスを記述する際は、回路の役割に合った回路基底クラスをクラスライブラリの中から選択して継承することにより、回路モジュールのクラスを回路基底クラスの派生クラス（実回路クラスあるいはユーザ作成クラス）として定義する。そして、それぞれの関数をオーバーライドすることにより、所望の動作をする回路クラスを実現する。

【００３９】

例えば、実回路クラスの例としては、図６のようなクラスが挙げられる。すなわち、ＰＣＩ（Peripheral Component Interconnect）バスをモデル化する際に用いられる３つのクラスのなかで、クラスCmBusMstIntfを回路基底クラスとするクラスCmPciBusMstIntf、クラスCmBusSlvIntfを回路基底クラスとするクラスCmPciBusMstIntfおよびクラスCmBusSystemを回路基底クラスとするクラスCmPciBusMstIntf、Ｖ８５０（商標）ＣＰＵをモデル化する際に用いられるクラスCmCpuを回路基底クラスとするクラスCmV850などが一例として挙げられる。

【００４０】

システムに含まれる各回路クラスを作成した後は、これらを含むシステムを１つのクラスとして記述する。このシステムのクラスには、上で作成した各回路クラスのオブジェクトが含まれる。また回路情報表示部もシステムのクラスに組み込む。このシステムのクラスも各回路クラスと同様に、システムの性質に合った回路基底クラスをクラスライブラリの中から選んで、継承する。

【００４１】

上記本発明に基づくシミュレーションの方式は、以下のようになる。図７に一例を示す。

【００４２】

１．シミュレーション対象となるシステムをＣ＋＋言語により記述する。

【００４３】

（A’）各回路モジュールのクラスを宣言（作成）する。ここで、各回路モジュールのクラスは、あらかじめ作成しておいた複数の回路基底クラスからなるクラスライブラリ３００の中からそれぞれに適した回路基底クラスを継承することによって宣言する。図７の例では、ユーザ作成ＣＰＵクラス１０１をＣＰＵ回路基底クラス（クラスCmCpu）３０１を継承することで宣言し、ユーザ作成バスマスタインターフェースクラス１０２をバスマスタインターフェース回路基底クラス（クラスCmBusMstIntf）３０２を継承することで宣言し、そして、ユーザ作成バスクラス１０３をバス回路基底クラス（クラスCmBusSystem）３０３を継承することで宣言している。

【００４４】

（B’）次に、システムのクラスを作成する。ここで、システムのクラスも、クラスライブラリ３００の中からそれに適した回路基底クラスを継承することによって作成する。図７の例では、同期回路回路基底クラス（クラスCmSyncModule）３０４を継承することで作成している（図７の②’）。

【００４５】

（C’）次に、回路の情報表示部（ＣＰＵ表示部１０５およびバス情報表示部１０６）を組み込む。

【００４６】

なお、上記各手順（A’），（B’），（C’）は、クラスライブラリ３００に定義されている回路基底クラスを継承することで各回路モジュールを示すクラスおよび複数の回路モジュールからなるシステムのクラスを作成する点を除いて、基本的な内容は、図２を参照して上記［前提事項］の欄で述べた手順（A），（B），（C）と同様である。また、図７では、図２に示すものに対応する構成に、同一の符号を付けている。

【００４７】

２．次に、シミュレーション制御部１０７を組み込み、シミュレータソース１０８を作成する。

【００４８】

３．項目２．の記述をＣ＋＋コンパイラ１０９でコンパイルして、シミュレータ

の実行形式１１０を得る。

【００４９】

４．これを実行することにより、シミュレーションを行う。

【００５０】

本実施の形態では、システムが回路基底クラスを継承していることにより、システムも所定の回路モジュールの性質を持つ。したがって、あるシステムを、別のシステムに含まれる回路モジュールとして用いることが可能になる。つまり、階層的にシステムを記述することが可能である。階層的にシステムを記述した場合についても本発明の効果が得られる。

【００５１】

[回路基底クラス詳細]

【００５２】

次に、図４に示す回路基底クラスについて詳細に説明する。

【００５３】

（１）CmComponentクラスは、以下の関数を持ち、クロックを持たない組合せ回路を記述する際に用いられる。

【００５４】

・Reset関数を持つ。これは回路の非同期リセットに用いられる。

【００５５】

・GetPinValue関数を持つ。これは他の回路Ｘからこの回路Ｙの端子の信号値を読み取る際に、回路Ｘによって実行される。

【００５６】

・SetPinValue関数を持つ。これは他の回路Ｘからこの回路Ｙの端子に信号を印加する際に、回路Ｘによって実行される。

【００５７】

Reset関数は、純粋仮想関数である。よって、この関数はCmComponentを継承したクラスで処理の内容を定義しなくてはならない。また、本実施の形態では、各回路間で端子を介して行う通信のほかに、バスを介しての通信を行うことができるようになっている。このバスを介して通信を行う場合については、CmBusMaste

rクラスやCmBusSlaveクラスを用いて高速にシミュレーションを行うようにする。したがって、バスを介しない通信は上記GetPinValue関数とSetPinValue関数を用いて記述することになる。

【００５８】

（２）CmSyncModuleクラスは、以下の関数を持ち、クロックを持つ順序回路を記述する際に用いられる。

【００５９】

・Reset関数、GetPinValue関数、SetPinValue関数を持つ。これらの意味はクラスCmComponentに同じである。

【００６０】

・OneStep関数を持つ。これは回路を1ステップ動作させるために用いられる。OneStep関数は、純粋仮想関数である。よって、この関数はCmSyncModuleを継承したクラスで処理の内容を定義しなくてはならない。

【００６１】

（３）CmBusMasterクラスは、以下の関数を持ち、バスマスタのメイン部を記述するときに使用される。ここでバスマスタとは、バスを能動的に駆動する回路モジュールを記述するときに使用される。また下記説明文中で参照されるバスマスタインターフェースとは、バスの差違を吸収するためにバスとバスマスタの間に挿入される回路モジュールである。

【００６２】

・Reset関数、GetPinValue関数、SetPinValue関数、OneStep関数を持つ。これらの意味はクラスCmSyncModuleに同じである。

【００６３】

・ReadBusMaster関数を持つ。これはこのオブジェクトからバスへ値を読み出す。

【００６４】

・WriteBusMaster関数を持つ。これはこのオブジェクトへバスから来た値を書き込む。

【００６５】

・ConnectBusMstIntf関数を持つ。これはバスマスタインターフェースを接続するために用いられる。

【００６６】

ReadBusMaster関数、WriteBusMaster関数等は、純粋仮想関数である。よって、これらの関数はクラスCmBusMasterを継承したクラスで処理の内容を定義しなくてはならない。

【００６７】

（４）CmBusSlaveクラスは、以下の関数を持ち、バススレーブのメイン部を記述するときに使用される。ここでバススレーブとは、バスから受動的にデータを受け取る回路モジュールである。また下記説明文中で、バススレーブインターフェースとは、バスの差違を吸収するためにバスとバススレーブの間に挿入される回路モジュールである。

【００６８】

・Reset関数、GetPinValue関数、SetPinValue関数、OneStep関数を持つ。これらの意味はクラスCmSyncModuleに同じである。

【００６９】

・ReadBusSlave関数を持つ。これは、このオブジェクトからバスへ値を読み出す。

【００７０】

・WriteBusSlave関数を持つ。これは、このオブジェクトへバスから来た値を書き込む。

【００７１】

・ConnectBusSlvIntf関数を持つ。これはバススレーブインターフェースを接続するために用いられる。

【００７２】

ReadBusSlave関数、WriteBusSlave関数等は、純粋仮想関数である。よって、これらの関数はクラスCmBusSlaveを継承したクラスで処理の内容を定義しなくてはならない。

【００７３】

（５）CmBusMstIntfクラスは、以下の関数を持ち、バスマスタインターフェースを作成する際に使用される。ここで、バスマスタインターフェースとは、バスを能動的に駆動する際の、バスの差違を吸収する回路モジュールである。

【００７４】

・Reset関数、GetPinValue関数、SetPinValue関数、OneStep関数を持つ。これらの意味はクラスCmSyncModuleに同じである。

【００７５】

・IsBusReq関数を持つ。これは、このバスマスタインターフェースがバスの使用を要求しているかどうかを戻す。

【００７６】

・AssertBusGnt関数を持つ。これは、外部の回路（通常アービタ）からこのバスマスタインターフェースに対し、バスの使用を許可することを通知する。

【００７７】

・DiassertBusGnt関数を持つ。これは、外部の回路（通常アービタ）からこのバスマスタインターフェースに対し、バスを使用を許可しないことを通知する。

【００７８】

・AppendWriteData関数、StartWrite関数、IsWriteEnd関数、AppendReadAddress関数、StartRead関数、IsReadEnd関数、GetReadData関数を持つ。これらはクラスCmBusMasterオブジェクトからクラスCmBusSystemオブジェクトを制御するために使用される。

【００７９】

AppendWriteData関数は、データを書き込むべきバススレーブのアドレス、書き込むべきデータ、書き込むべきデータのバイト数を当該クラス内の所定の変数領域（バッファ）に書き込む関数である。StartWrite関数は、バスへのデータ等の書き込み動作を開始する関数である。IsWriteEnd関数は、バスへのデータ等の書き込み動作が終了したか否かの結果を戻す関数である。AppendReadAddress関数は、データを読み出すべきバススレーブのアドレス、読み出すデータのバイト数を上記バッファに書き込む関数である。StartRead関数は、バスからのデータの読み出し動作を開始する関数である。IsReadEnd関数は、バスからのデータの

読み出し動作が終了したか否かの結果を戻す関数である。GetReadData関数は、バススレーブから上記バッファに読み出したデータをそのバッファから取り出す関数である。

【００８０】

（６）CmBusSlvIntfクラスは、以下の関数を持ち、バスから受動的にデータを受け取る際の、バスの差違を吸収する回路モジュールを記述するために使用される。

【００８１】

・Reset関数、GetPinValue関数、SetPinValue関数、OneStep関数を持つ。これらの意味はクラスCmSyncModuleに同じである。

【００８２】

・IsBusReq関数を持つ。これは、このバススレーブインターフェースがバスの使用を要求しているかどうかを戻す。

【００８３】

・AssertBusGnt関数を持つ。これは、外部の回路（通常アービタ）からこのバススレーブインターフェースに対し、バスの使用を許可することを通知する。

【００８４】

・DiassertBusGnt関数を持つ。これは、外部の回路（通常アービタ）からこのバススレーブインターフェースに対し、バスを使用を許可しないことを通知する。

【００８５】

・AppendWriteData関数、StartWrite関数、IsWriteEnd関数、AppendReadAddress関数、StartRead関数、IsReadEnd関数、GetReadData関数を持つ。クラスCmBusSystemの派生クラスからクラスCmBusSlvIntfへ値を読み書きするために使用される。読み書きの対象が異なることを除いて、これらの関数における基本的な機能はクラスCmBusMstIntfの場合と同様である。

【００８６】

（７）CmBusSystemクラスは、以下の関数を持ち、データをマスタからスレーブに伝達する回路モジュールを記述するときに使用される。

【００８７】

・Reset関数、GetPinValue関数、SetPinValue関数、OneStep関数を持つ。これらの意味はクラスCmSyncModuleに同じである。

【００８８】

・ConnectBusMstIntf関数を持つ。これはクラスCmBusMstIntfの派生クラスをこのオブジェクトに接続する。

【００８９】

・ConnectBusSlvIntf関数を持つ。これはクラスCmBusSlvIntfの派生クラスをこのオブジェクトに接続する。

【００９０】

・SetUserReadMap関数を持つ。これは、このバスのバスリードマップを指定する際に使用される。バスリードマップとは、バスを介して回路モジュールから値を読み出すときに使われるアドレスマップである。バスリードマップはCmHierクラスのUserReadMap関数として記述される。

【００９１】

・SetUserWriteMap関数を持つ。これは、このバスのバスライトマップを指定する際に使用される。バスライトマップとは、バスを介して回路モジュールへ値を書き込むときに使われるアドレスマップである。バスライトマップはCmHierクラスのUserWriteMap関数として記述される。

【００９２】

・SetUserArbitor関数を持つ。これは、このバスのアービタを指定する際に使用される。アービタとは、バスの使用権を調停する回路である。この回路はCmHierクラスのUserArbitor関数として記述される。

【００９３】

・ReadMap関数を持つ。これは、このバスを介して回路モジュールから情報を取り出す。

【００９４】

・WriteMap関数を持つ。これは、このバスを介して回路モジュールへ情報を書き込む。

【００９５】

（８）CmCpuクラスは、クラスCmBusMasterを継承しており、以下の関数を持ち、ＣＰＵを記述するときに使用される。

【００９６】

・Reset関数、GetPinValue関数、SetPinValue関数、OneStep関数、ConnectBusMstIntf関数を持つ。これらの意味はクラスCmSyncModuleに同じである。

【００９７】

・ConnectBusMstIntf関数を持つ。これはバスマスタインタフェースを接続するために用いられる。

【００９８】

・GetPc関数を持つ。これは現在のプログラムカウンタの値を戻す。

【００９９】

・GetReg関数を持つ。これは指定されたレジスタの値を戻す。

【０１００】

・SetReg関数を持つ。これは指定されたレジスタの値を戻す。

【０１０１】

・GetData関数を持つ。これは指定されたアドレスに格納されたデータを戻す。

【０１０２】

・SetData関数を持つ。これは指定されたアドレスのデータを上書きする。

【０１０３】

以上のように、ＣＰＵをモデルとする回路基底クラスCmCpuにおいては、バスマスタをモデルとする回路基底クラスCmBusMasterを継承することで、クラスCmBusMasterで定義される関数を持つほか、GetPc関数、GetReg関数、SetReg関数、GetData関数、およびSetData関数を持つことによって、プログラム実行時における各レジスタやプログラムカウンタの情報を取り出してＣＰＵ情報表示部によって表示すること等が可能となる。

【０１０４】

（９）CmMemoryクラスは、クラスCmBusSlaveを継承しており、以下の関数を持ち、メモリを記述するときに使用される。

【０１０５】

・Reset関数、GetPinValue関数、SetPinValue関数、OneStep関数、ConnectBusSlvIntf関数を持つ。これらの意味はクラスCmBusSlaveに同じである。

【０１０６】

・GetData関数を持つ。これは指定されたアドレスに格納されたデータを戻す。

【０１０７】

・SetData関数を持つ。これは指定されたアドレスのデータを上書きする。

【０１０８】

（１０）CmHierクラスは、クラスCmSyncModuleを継承しており、以下の関数を持ち、バスを含むシステムを記述するときに使用される。

【０１０９】

・Reset関数、GetPinValue関数、SetPinValue関数、OneStep関数を持つ。これらの意味はCmBusSlaveに同じである。

【０１１０】

・UserReadMap関数を持つ。これは、このシステムに含まれるバスのバスリードマップを記述する際に使用される。ここで記述されたバスリードマップはCmBusSystemクラスのSetUserReadMap関数により、バスに登録される。

【０１１１】

・UserWriteMap関数を持つ。これは、このシステムに含まれるバスのバスライトマップを記述する際に使用される。ここで記述されたバスライトマップはCmBusSystemクラスのSetUserWriteMap関数により、バスに登録される。

【０１１２】

・UserArbitor関数を持つ。これは、このシステムに含まれるバスのアービタを記述する際に使用される。ここで記述されたアービタはCmBusSystemクラスのSetUserArbitor関数により、バスに登録される。

【０１１３】

次に、本実施の形態においてバスを介してデータを伝送する場合のシステムの記述例について説明する。ここで、システムの記述例を説明する前に上述したアドレスマップとアービタについて簡単に説明する。

【０１１４】

アドレスマップとは、どのアドレスに対してどの回路が割り当てられているのかを示した表である。アービタとは、複数のバスマスタがバスを使用したいと要求したときに、どのバスマスタに使用権を与えるかを調停する回路である。これらは、一般に良く知られているものである。

【０１１５】

まず、アドレスマップについて説明する。バスにはたくさんの回路が接続されている。そのため、バスにただデータを流しただけでは、どの回路に対する指示なのかを判断することができない。そこで、回路を指定するためにアドレスを用いている。バスには、データとアドレスが必ず対にして流される。また、アドレスがどのような値のときはデータをどの回路に渡すかということを決めておく。このアドレスと回路の対応関係がアドレスマップである。

【０１１６】

バスにデータとアドレスを対にして流すことは、本実施の形態においては、クラスCmHierのUserReadMap関数およびUserWriteMap関数にデータとアドレスを対にして渡すことに対応している。アドレスマップは、本実施の形態においては、UserReadMap関数およびUserWriteMap関数内に、渡されたアドレスに対応する回路を判断し、その回路に対してデータを渡す、という処理を書くことに対応している。

【０１１７】

一方、アービタは、２つ以上のバスマスタがバスに接続されているときに必要になる。バスはただの信号線に信号を流すだけの仕組みである。よって、２つ以上のバスマスタが同時にバスに信号を流すと、２つの信号が１つの信号線で衝突することになる。よって、正しい値を目的の回路モジュールに伝えることができなくなる。これを避けるため、アービタというものが使われている。一般的に、アービタを使うときは、バスマスタがバスに信号を流す前に、必ずアービタにバスの使用を要求することになる。アービタは各バスマスタのから要求を見て、どれか一つのバスマスタにだけバスの使用を許可する。バスマスタはバス使用許可が来てはじめて、バスを使用できるようになる。

【０１１８】

本実施の形態では、バスマスタを、バスマスタとバスマスタインターフェースに分割して構成している。したがって、バスの使用要求/許可はアービタとバスマスタインターフェースの間で行われる。

【０１１９】

以下、図８～図１３を参照して、バスを介したデータの通信例について説明する。図８は、この例で用いるシステムＡＢＣの構成を示すブロック図であり、図９～図１３は、Ｃ＋＋言語によるシステムの記述例を示す一連のリストであって、図９～図１３の順に連続している。

【０１２０】

図８に示すように、この例では、システムＡＢＣを、ＰＣＩバス４０１を介して、２つのＣＰＵ（ＣＰＵ１（４０２）、ＣＰＵ２（４０３））と、タイマ１（４０４）と、カウンタ１（４０５）とから構成している。ＣＰＵ１（４０２）、ＣＰＵ２（４０３）、タイマ１（４０４）、およびカウンタ１（４０５）にはそれぞれＰＣＩバス４０１とのインタフェースとなるインタフェース４０２ａ、４０３ａ、４０４ａ、および４０５ａを設けている。なお、システムの記述例を説明する際には、図８のシステムＡＢＣに、図示していない他の回路が複数接続されているものとしてそれらの回路に関する記述の記述位置についても説明する。

【０１２１】

システム記述においては、まず、図９に示すように、システムＡＢＣに対応するオブジェクトABCをクラスCmHierを継承するものとして宣言している（図９の①）。次に、図８のＰＣＩバス（４０１）に対応するＰＣＩバスを、システムABCの中にユーザ作成クラス（実回路クラス）CmPciBusSystem（回路基底クラスはクラスCmBusSystemであるとする。）として作成する（図９の②）。次に、図８のＣＰＵ１（４０２）、ＣＰＵ２（４０３）に対応するものとして、システムABCの中にオブジェクトCpu1，Cpu2というクラスCmCpuによる２つのＣＰＵを作成する（図９の③）。また、オブジェクトCpu1，Cpu2はバスマスタなので、システムABCの中に、バスマスタとしてのオブジェクトCpu1，Cpu2をバスに繋ぐためのバスマスタインターフェースオブジェクトCpu1BusMstIntf，Cpu2BusMstIntfを作成する（図９の④；図８のインターフェース４０２ａ，４０３ａに対応）。このと

き、バスマスタインターフェースCpu1BusMstIntf，Cpu2BusMstIntfは、ユーザ作成クラスでCmPciBusMstIntf（クラスCmPciBusMstIntfの回路基底クラスはクラスCmBusMstIntfであるとする。）によって作成される。

【０１２２】

次に、システムABCの中にタイマ１（４０４）に対応するTimer1という回路を作成する（図９の⑤）。また、オブジェクトTimer1はバススレーブなので、システムABCの中に、オブジェクトTimer1をバスに繋ぐためのバススレーブインターフェース（図８のインターフェース４０４ａに対応）を作成する（図９の⑥）。次に、システムABCの中にカウンタ１（４０５）に対応するCounter1という回路を作成する（図９の⑦）。また、オブジェクトCounter1はバススレーブなので、システムABCの中に、オブジェクトCounter1をバスに繋ぐためのバススレーブインターフェース（図８のインターフェース４０５ａに対応）を作成する。ここで、オブジェクトTimer1は、ユーザ作成クラスCmTimer1に基づくものとして、オブジェクトCounter1は、ユーザ作成クラスCmCounter1に基づくものとして作成されるが、ユーザ作成クラスCmTimer1，CmCounter1は、ともにクラスCmBusSlaveの派生クラスであるとする。また、それぞれのバススレーブインターフェースオブジェクトTimer1BusSlvIntf，Counter1BusSlvIntfは、ともにクラスCmBusSlvIntfの派生クラスCmPciBusSlvIntfに基づいて宣言されている。

【０１２３】

また、他のモジュールについての記述は図９の⑨の位置に記述する。

【０１２４】

次に、UserBusReadMap関数について記述する（図１０の①）。UserBusReadMap関数では、バスリードマップを記述する。戻り値は、RDATAという構造体である。RDATA.Statusは、０なら読みだし成功、１なら読みだし失敗を意味する。RDATA.Dataは、読みだし成功のときだけ、読み出した値を戻す。また、ULONGは32bitの信号値を意味する。UserBusReadMap関数は、アドレスaddressの値を元に、値を読み出す回路を決めて、その回路のバススレーブインターフェースの読みだし関数を実行する。まず、戻り値を持つ変数vを構造体RDATAとして定義する（図１０の②）。

【０１２５】

次に、アドレス100から200までは、回路（オブジェクト）Timer1から値を読み出すことに設定するための判定式を記述する（図１０の③）。判定式が真の場合に実行されるものとして、オブジェクトTimer1BusSlvIntfに対してAppendReadAddress関数を0，address-100，byte_countの組を引数として実行することによって、読み出すアドレスとバイト数を指定する（図１０の④）。次に、オブジェクトTimer1BusSlvIntfに対してStartRead関数を実行して、値を読み出している（図１０の⑤）。次に、IsReadEnd関数によって、読み出しが成功したかどうかを調べ（図１０の⑥）、読みだしが成功した場合には、v.Statusを０に設定して（図１０の⑦）、オブジェクトTimer1BusSlvIntfに対してGetReadData関数を実行することで、読み出した値を取り出し、v.Dataに設定している（図１０の⑧）。一方、読み出しが失敗した場合には、v.Statusに１を設定して復帰するようにしている（図１０の⑨）。ここでAppendReadAddress関数、StartRead関数、IsReadEnd関数およびGetReadData関数を実行する場合には、第１引数には0を指定して、これらの関数の実行が通常の転送方法によるものであることを通知する。この第１引数は、後述するマスタとスレーブを入れ替えた転送方式（Split転送）と、通常の転送方法とを区別するために用意されている。

【０１２６】

また、回路Counter1に対しては、読み出しアドレスを200から300までとして、回路Timer1のときと同様の記述を行う（図１０の(10)～(11)）。さらに、他の回路に関して同様に記述する（図１０の(12)）。

【０１２７】

次に、UserBusWriteMap関数によって、バスライトマップを記述する（図１１の①）。戻り値は、int（整数）であり、０なら書き込み成功、１なら書き込み失敗を意味することとする。また、引数addressとdataは、ともにULONG（32bit）の信号値である。

【０１２８】

ここでは、アドレスaddressの値を元に、値を書き込む回路を決めて、その回路のバススレーブインターフェースの書き込み関数を実行する（図１１の②～）

。まず、アドレス100から200までは、回路Timer1へ値を書き込むことにする（図１１の③）。オブジェクトTimer1BusSlvIntfに対してAppendWriteData関数を0, (address-100), data, byte_countの組を引数として実行し、書き込む情報を指定する（図１１の④）。次に、StartWrite関数を実行して値を書き込む（図１１の⑤）。次に、回路Timer1に対してIsWriteEnd関数を実行することで、書き込みが成功したかどうかを調べる（図１１の⑥）。書き込みが成功した場合には、戻り値を０にし、書き込みが失敗した場合には、戻り値を１にして復帰する。ここでAppendWriteData関数、StartWrite関数、およびIsWriteEnd関数を実行する場合には、上述したAppendReadAddress関数などを実行する場合と同様に、第１引数には０を指定して、これらの関数の実行が通常の転送方法によるものであることを通知する。

【０１２９】

次に、回路Counter1について記述する。回路Counter1については、アドレス250から350までとして、回路Timer1のときと同様に記述する（図１１の⑨～(10)）。また、他の回路についても同様に記載する（図１１の(11)）。

【０１３０】

次に、関数UserArbitorおいてアービタに関する動作を記述する（図１２の①）。関数UserArbitorの戻り値はvoid（なし）である。なお、バス使用許可はバスマスタインターフェースに直に与えることとする。まず、オブジェクトCpu1BusMstIntfおよびオブジェクトCpu2BusMstIntfに対してIsBusReq関数を実行し、バス使用要求をバスマスタインターフェースから直に取り出す（図１２の②）。次に、ここではオブジェクトCpu1がオブジェクトCpu2よりも優先するとして、判定を行い（図１２の③）、Cpu1がバスを要求していたら、Cpu2がバスを要求しているかどうかに関わらず、Cpu1にバスを使用させる。すなわち、オブジェクトCpu1BusMstIntfに対してAssertBusGnt関数を実行し、続いてオブジェクトCpu2BusMstIntfに対してDiassertBusGnt関数を実行して、Cpu1を使用可、Cpu2を使用不可とする（図１２の④）。

【０１３１】

一方、オブジェクトCpu1がバスを使用していない場合で、かつ、オブジェクト

Cpu2がバスを使用しようとしているときに（図１２の⑤）、Cpu1を使用不可にして、Cpu2を使用可とする（図１２の⑥）。また、両者から使用要求が出されていない場合には、両者を使用不可に設定する（図１２の⑦）。

【０１３２】

次に、システムABCの接続および動作を関数ABC(void)内に記述する（図１３の①）。関数ABC(void)は、Ｃ＋＋言語の仕様によって、システムABCのオブジェクトを作る時に呼ばれる関数（コンストラクタ；クラス名と同一名称の関数）である。まず、各回路と各バスインターフェースを、ConnectBusMstIntf関数またはConnectBusSlvIntf関数によって接続するとともに、各バスインターフェースとＰＣＩバスとをConnectBusMstIntf関数によって接続する（図１３の②）。このとき、各回路のバスインターフェースオブジェクトのアドレスを引数とすることによって、ＰＣＩバスと各回路が接続される。また、他の回路についても同様に記述する（図１３の③）。

【０１３３】

次に、オブジェクトPciBusSystemに対してSetUserReadMapおよびSetUserWriteMap関数を実行することで、アドレスマップを指定する（図１３の④）。この関数を実行することによって、上述したアドレスマップが、オブジェクトPciBusSystemに登録されることになる。このことで、PciBusSystemは、上述したアドレスマップを使用できるようになる。

【０１３４】

次に関数OneStepによって、システムABCが１単位時間にどのような動作をするかを記述する（図１３の⑤）。システムABCは、システムのそれぞれの構成要素が勝手に動くことによって動作する。したがって、システムABCの階層ですることと言えば、それぞれのシステムの構成要素を１単位時間動作させることだけである（図１３の⑥（コメント文））。まず、ＰＣＩバスPciBusSystemに対して、OneStep関数を実行する（図１２の⑦）。バスは信号線なので、本来はクロックは必要ないが、デバグ機能として、クロックに同期して情報を取り出せるようにするため、クロックを与えている。次に、各オブジェクトCpu1BusMstIntf, Cpu2BusMstIntf, Timer1BusSlvIntf, Counter1BusSlvIntf, Cpu1, Cpu2, Timer1, Coun

ter1に対してOneStep関数を実行して１単位時間の動作を実行させる（図１２の⑧）。また、その他の回路についても同様にOneStep関数を動作させる記述を行う（図１２の⑨）。

【０１３５】

以上の記述によってシステムABCのシステム記述が完成する。本実施の形態では、この例のように、各モジュール間を接続するバスを、バスシステムと、バスマスタ・インターフェースまたはバススレーブ・インターフェースとに分割して記述するとともに、各オブジェクト（回路モジュール）をあらかじめ定めた継承関係に基づいて記述することで、システムにおける各オブジェクト間の接続インターフェースを統一化された形態で記述することができる。

【０１３６】

[回路基底クラスのヘッダファイル]

【０１３７】

次に、上述した各回路基底クラスのうちの主なもののヘッダファイルについて図１４～図２３を参照して説明する。なお、各図においては、符号“//”を付したコメント行によって各記述の詳細な説明を行っている。なお、記述中“virtual void Reset(void) = 0;”（例えば図１４）のように、“virtual”と“=0”を付加した関数は、純粋仮想関数であり、そのクラスを継承するクラスにおいてその関数の定義が必ず必要であることを示している。また、各図では、理解をしやすくするため、各関数の詳細な記述を省略してコメント行のみとしている。各図の内容は次の通りである。

【０１３８】

図１４は、クラスCmComponentを定義する記述である。クラスCmComponentでは、Reset関数を純粋仮想関数として定義している。図１５は、クラスCmComponentの派生クラスであるクラスCmSyncModuleを定義する記述である。クラスCmComponentでは、OneStep関数を純粋仮想関数として定義している。なお、クラスCmSyncModuleは、クラスCmComponentの派生クラスであり、またクラスCmComponent内の関数のオーバーライドを行っていないので、クラスCmComponentの関数についての定義はそのまま取り込まれることになる。

【０１３９】

図１６は、クラスCmSyncModuleの派生クラスであるクラスCmBusMasterを定義する記述を示している。図１７は、クラスCmSyncModuleの派生クラスであるクラスCmBusSlaveを定義する記述を示している。図１８および図１９は、クラスCmSyncModuleの派生クラスであるクラスCmBusMstIntfを定義する記述を示している。クラスCmBusMstIntfの定義では、図１９に示す“virtual void OneStep(void) {…}”という仮想関数によって、この回路を１ステップ動作させるための記述をクラスCmSyncModuleで定義されたOneStep関数をオーバーライドすることで記述している。また、図１８のAppendWriteData関数、StartWrite関数、IsWriteEnd関数、およびAppendReadAddress関数、ならびに、図１９のStartRead関数、IsReadEnd関数、およびGetReadData関数の引数として、整数変数modeを設定することで、後述するマスタとスレーブを入れ替えた転送方法（Split（スピリット）転送）の動作を可能としている。図２０および図２１は、クラスCmSyncModuleの派生クラスであるクラスCmBusSlvIntfを定義する記述を示している。図２２は、クラスCmSyncModuleの派生クラスであるクラスCmBusSystemを定義する記述を示している。そして、図２３は、クラスCmSyncModuleの派生クラスであるクラスCmHierを定義する記述を示している。

【０１４０】

[バスシミュレーション方式]

次にバスシミュレーションの方式すなわち具体的なシミュレーションの動作について詳細に説明する。バスシミュレーションの基本的な方針は、次のようになっている。

【０１４１】

●バスマスタはバスの細かい制御をバスマスタインターフェースに任せる。バスマスタは読み書きすべきデータをすべて一気に瞬間にバスマスタインターフェース内のバッファに登録する。

【０１４２】

●バスマスタインターフェースからバススレーブまでの通信は、OneStep関数が実行されるたびに実行される。

【０１４３】

・バスマスタインターフェースはOneStepが呼ばれるたびにデータを1つずつバッファから取り出し、バスへ渡す。

【０１４４】

・バスはバスリードマップ/バスライトマップから、読み書きの対象となるバススレーブを判定し、そのバススレーブに対応するバススレーブインターフェースに読み書きの指示を出す。

【０１４５】

・バススレーブインターフェースはバスから来た指示に基づきバススレーブを読み書きし、結果をバスに戻す。

【０１４６】

・バスはバススレーブインターフェースから戻された結果をバスマスタインターフェースに返す。

【０１４７】

・バスマスタインターフェースはバスから戻された結果を元に、次のOneStepでの動作を決める。

【０１４８】

次に、上記の実施形態における具体的な関数実行の流れを、データの書き込みの例で説明する。例えば、図２４に示すシステムXにおいて、バスマスタA1、バスマスタインターフェースB1、バスC1、バススレーブインターフェースD1、バススレーブE1がこの順に接続されているものとする。このシステムにおいて、例えば、図２５に示す２組の情報（アドレス/データ/バイト数…0x00100000/0x00112233/4，0x00100004/0x44556677/4）を、例えばＣＰＵ等の回路であるバスマスタA1から、例えばメモリ等の回路であるバススレーブE1に書き込むことにする。

【０１４９】

なお、システムXならびに、バスマスタA1、バスマスタインターフェースB1、バスC1、バススレーブインターフェースD1、およびバススレーブE1に関する記述は、上述した対応する各クラスの性質を継承して記述されているものとする。すなわち、システムXはクラスCmSyncModuleを、バスマスタA1はクラスCmBusMaster

を、バスマスタインターフェースB1はクラスCmBusMstIntfを、バスC1はクラスCmBusSystem、バススレーブインターフェースD1はクラスCmBusSlvIntfを、そして、バススレーブE1はクラスCmBusSlaveを、それぞれ直接あるいはそれらをさらに継承するクラスを継承して作成されているものとする。また、システムXのUserWriteMap関数により、アドレス0x00100000および0x00100004はバススレーブインターフェースD1に対応づけられているものとする。また、システムXの構築時に、バスC1のSetUserWriteMap関数を実行することにより、システムXのUserWriteMap関数がバスC1のバスライトマップとしてすでに登録されているものとする。また、いつも、バスマスタA1、バスマスタインターフェースB1、バスC1、バススレーブインターフェースD1、およびバススレーブE1それぞれに対しては、それぞれを１ステップ動作させる関数OneStepが、システムXのOneStepの中から呼び出されていることとする。

【０１５０】

以上の構成において、図２６に示すような単位時間ごとの各時間１～４におけるシステムXの動作を図２７～図３０を参照して説明する。

【０１５１】

[時間１]

【０１５２】

まず、時間１におけるシステムXの動作を図２７を参照して説明する。図２７は、同一単位時間（時間１）内における各回路の動作を時系列で示す図である。時間１では、まず、バスマスタA1のOneStep関数が実行される。そこでは、バスマスタA1が、バスマスタインターフェースB1の関数AppendWriteData(0,0x00100000,0x00112233,4)を実行する。次に、バスマスタインターフェースB1は情報をバッファに保存しておく。次に、バスマスタA1はバスマスタインターフェースB1のAppendWriteData(0,0x00100004,0x44556677,4)関数を実行する。次に、バスマスタインターフェースバスマスタインターフェースB1は情報をバッファに保存しておく。次に、バスマスタA1はバスマスタインターフェースB1のStartWrite関数を実行する。次に、バスマスタインターフェースB1は書き込みが要求されたことを覚えておく。次に、バスマスタA1はバスマスタインターフェースB1のIsWriteEnd

関数を実行する。次に、バスマスタインターフェースB1は、バスマスタA1に対してまだ書き込み動作中であることを戻す。次に、バスマスタA1は待機状態となる。

【０１５３】

バスマスタA1のOneStep関数が終了すると、次に、バスマスタインターフェースB1のOneStep関数が実行される。そこでは、バスマスタインターフェースB1はバスマスタA1から書き込みが要求されたので、バスの使用を要求する。これは次に外部からバスマスタインターフェースB1のIsBusReq関数が実行されたときの戻り値を真にすることに対応する。ここで、バスマスタインターフェースB1のOneStep関数が終了する。

【０１５４】

バスマスタインターフェースB1のOneStep関数が終了すると、次に、バスC1のOneStep関数が実行される。バスC1のOneStep関数は、まず、アービタを動作させる。次に、アービタはバスマスタインターフェースB1のIsBusReq関数を実行する。次に、バスマスタインターフェースB1はバスの使用を要求していることを戻す。ここで、アービタはバスマスタインターフェースB1にバスの使用を許可したとする。アービタはバスマスタインターフェースB1のAssertBusGnt関数を実行する。バスマスタインターフェースB1はバスを許可されたことを覚えておく。ここで、バスC1のOneStep関数が終了する。

【０１５５】

バスC1のOneStep関数が終了すると、次に、バススレーブインターフェースD1のOneStep関数が実行される。時間1では、バススレーブインターフェースD1のOneStep関数は、何も行わずに終了する。

【０１５６】

バススレーブインターフェースD1のOneStep関数が終了すると、次に、バススレーブE1のOneStep関数が実行される。ここでは、バススレーブE1固有の動作をするが、バスに関しては何も動作しない。

【０１５７】

[時間２]

【０１５８】

次に、次の単位時間である時間2におけるシステムXの動作について図２８を参照して説明する。時間2では、時間1の場合と同様に、まず、バスマスタA1のOneStep関数が実行される。ここでは、バスマスタA1はバスマスタインターフェースB1のIsWriteEnd関数を実行する。この場合、バスマスタインターフェースB1はまだ書き込み動作中であることを戻す。次に、バスマスタA1は待機状態となる。

【０１５９】

バスマスタA1のOneStep関数が終了すると、バスマスタインターフェースB1のOneStep関数が実行される。バスマスタインターフェースB1のOneStep関数では、バスマスタインターフェースB1はバスの使用を許可されたので、バスに書き込む。つまり、次の動作を行う。バスマスタインターフェースB1はバッファから情報を一つ取り出す。このとき、アドレス:0x00100000, データ:0x00112233, バイト数:4である。次に、バスマスタインターフェースB1はバスC1のWriteMap(0x00100000,0x00112233,4)関数を実行する。

【０１６０】

次に、バスC1のWriteMap関数は、以下の動作を行う。バススレーブインターフェースD1のAppendWriteData(0,0x00100000,0x00112233,4)関数を実行する。バススレーブインターフェースD1は与えられた情報をバッファに蓄える。バススレーブインターフェースD1のStartWrite関数を実行する。バススレーブインターフェースD1はバススレーブE1のWriteBusSlave(0x00100000,0x00112233,4)関数を実行する。バススレーブE1は与えられた情報を内部に書き込む。書き込みが成功したかどうかをバススレーブインターフェースD1に返す。ここでは成功したことにする。バススレーブインターフェースD1のIsWriteEnd関数を実行する。バススレーブインターフェースD1からバススレーブE1への書き込みは成功したので、真となる。バススレーブインターフェースD1のIsWriteEnd関数の戻り値を戻す。

【０１６１】

次に、バスマスタインターフェースB1は戻り値から、情報が正常に書き込まれたことを知る。そこで、バッファの中身を1つ削除する。バスマスタインターフェースB1にはまだバッファに1つ情報があるので、次のバスマスタインターフェ

ースB1のOneStepの実行でも同じことを行う。

【０１６２】

バスマスタインターフェースB1のOneStep関数が終了すると、バスC1のOneStep関数が実行される。時間2におけるバスC1のOneStep関数では、何も実行されない。

【０１６３】

次に、バスC1のOneStep関数が終了すると、バススレーブインターフェースD1のOneStep関数が実行される。時間2におけるバススレーブインターフェースD1のOneStep関数では、何も実行されない。

【０１６４】

次に、バススレーブインターフェースD1のOneStep関数が終了すると、バススレーブE1のOneStep関数が実行される。バススレーブE1のOneStep関数では、バススレーブE1固有の動作をするが、バスに関しては何も動作しない。

【０１６５】

[時間３]

【０１６６】

時間3では、図２９に示すように、まず、バスマスタA1のOneStep関数が実行される。バスマスタA1のOneStep関数では、バスマスタA1はバスマスタインターフェースB1のIsWriteEnd関数を実行する。この場合、バスマスタインターフェースB1はまだ書き込み動作中であることを戻す。ここで、バスマスタA1は待機状態となり、バスマスタA1のOneStep関数が終了する。

【０１６７】

バスマスタA1のOneStep関数が終了すると、バスマスタインターフェースB1のOneStep関数が実行される。時間3のバスマスタインターフェースB1のOneStep関数では、バスマスタインターフェースB1がバスの使用を許可されているので、バスに書き込む動作を行う。つまり、以下の動作を行う。

【０１６８】

バスマスタインターフェースB1はバッファから情報を一つ取り出す。取り出される情報は、アドレス:0x00100004，データ:0x44556677，バイト数:4の情報であ

る。次に、バスマスタインターフェースB1はバスC1のWriteMap(0x00100004,0x44556677,4)関数を実行する。

【０１６９】

ここで、バスC1のWriteMap関数は、以下を処理を行う。まず、バスC1のWriteMap関数は、バススレーブインターフェースD1のAppendWriteData(0,0x00100004,0x44556677,4)関数を実行する。ここで、バススレーブインターフェースD1は与えられた情報をバッファに蓄える。次に、バスC1のWriteMap関数は、バススレーブインターフェースD1のStartWrite関数を実行する。ここで、バススレーブインターフェースD1はバススレーブE1のWriteBusSlave(0x00100004,0x44556677,4)関数を実行する。ここで、バススレーブE1は与えられた情報を内部に書き込む。そして、書き込みが成功したかどうかをバススレーブインターフェースD1に返す。ここでは成功したことにする。次に、バスC1のWriteMap関数は、バススレーブインターフェースD1のIsWriteEnd関数を実行する。ここでは、バススレーブインターフェースD1からバススレーブE1への書き込みは成功したので、真となる。バスC1のWriteMap関数は、バススレーブインターフェースD1のIsWriteEnd関数の戻り値を戻す。

【０１７０】

バスマスタインターフェースB1は戻り値から、情報が正常に書き込まれたことを知る。そこで、バッファの中身を１つ削除する。バスマスタインターフェースB1のバッファには情報は無い。そこで、バスマスタインターフェースB1は書き込み動作を完了する。これは、次のバスマスタインターフェースB1のIsWriteEnd関数の戻り値を真とすることに対応する。ここで、バスマスタインターフェースB1のOneStep関数が終了する。

【０１７１】

バスマスタインターフェースB1のOneStep関数が終了すると、バスC1のOneStep関数が実行されるが、ここでは何も処理がなされずに、OneStep関数が終了する。

【０１７２】

バスC1のOneStep関数が終了すると、バススレーブインターフェースD1のOneSt

ep関数が実行されるが、ここでは何も処理がなされずに、関数が終了する。

【０１７３】

次に、バススレーブE1のOneStep関数が実行されて、バススレーブE1が固有の動作をする。ただし、バスに関しては何も処理は行われない。

【０１７４】

[時間４]

【０１７５】

次に、時間4になると、図３０に示すように、バスマスタA1のOneStep関数が実行される。時間4のバスマスタA1のOneStep関数では、バスマスタA1はバスマスタインターフェースB1のIsWriteEnd関数を実行する。ここで、バスマスタインターフェースB1は書き込みが終了したことを戻す。次に、バスマスタA1は次の動作を始める。時間4では、バスマスタインターフェースB1のOneStep関数、バスC1のOneStep関数、バススレーブインターフェースD1のOneStep関数が順次実行される。これらのOneStep関数では何も処理が行われず関数が終了する。次に、バススレーブE1のOneStep関数が実行される。バススレーブE1のOneStep関数では、バスに関しては何も処理が行われず、バススレーブE1固有の動作のみが行われる。

【０１７６】

以上の時間1～時間4の動作によって図２５に示すデータの書き込みが完了する。

【０１７７】

[Split（スピリット）転送について]

次に、バスを介したデータ転送の他の形態について説明する。通常バスは、バスマスタからバススレーブという方向だけデータの読み書きの指示が送られる。ところが高機能のバスには、バスマスタからバススレーブに対して送られた読み書きの要求に対して、バススレーブが即座に応答できない場合、一旦その通信を終了して、その後改めて、今度はバススレーブからバスマスタへ、通常とは逆の方向に、主体的に値を返すことがある。ここでは、このような状況をSplit転送と呼ぶことにする。Split転送とは、最初マスタからスレーブに指示が行った時点から、それが一旦終了して、逆にスレーブからマスタへ指示が行くまでの、一

連の動作を指す。Split転送は高機能なバスに存在することが知られている。本実施形態において、Split転送をシミュレーションする場合について説明する。

【０１７８】

Split転送では、上記の例と、スレーブとマスタの関係が全く逆になる。すなわち、

【０１７９】

●バススレーブはバスの細かい制御をバススレーブインターフェースに任せる。バススレーブは読み書きすべきデータをすべて一気に瞬間にバススレーブインターフェース内のバッファに登録する。

【０１８０】

●バススレーブインターフェースからバスマスタまでの通信は、OneStep関数が実行されるたびに実行される。ここでは、以下の処理が行われる。

【０１８１】

・バスマスタインターフェースはOneStep関数が呼ばれるたびにデータを1つずつバッファから取り出し、バスへ渡す。

【０１８２】

・バスはバスリードマップ/バスライトマップから、読み書きの対象となるバスマスタを判定し、そのバスマスタに対応するバスマスタインターフェースに読み書きの指示を出す。

【０１８３】

・バスマスタインターフェースはバスから来た指示に基づきバスマスタを読み書きし、結果をバスに戻す。

【０１８４】

・バスはバスマスタインターフェースから戻された結果をバススレーブインターフェースに返す。

【０１８５】

・バススレーブインターフェースはバスから戻された結果を元に、次のOneStep関数での動作を決める。

【０１８６】

これらを可能にするため、バススレーブとバスマスタの間で同じ機能が、バススレーブインターフェースとバスマスタインターフェースとで同じ機能があれば良いことになる。そこで、本実施の形態では、バススレーブとバスマスタ、バススレーブインターフェースとバスマスタインターフェースを同じ構造としている。また、現在どちらの方向に通信が行われているか（つまり、マスタ→スレーブかスレーブ→マスタか）を判定するために、AppendWriteDataなどの関数にはmodeという引数が用意されている（図１８、図１９、図２０、図２１参照）。このmodeが1のときは、通常と全く逆の動作をすることになっている。

【０１８７】

[回路クラスの作成例]

【０１８８】

次に、新たな回路クラスを作成する場合の具体例を説明する。ユーザが新たなバスマスタインターフェースクラスをCUserBusMstIntfという名前で作成する場合を考える。まず、新たなクラスCUserBusMstIntfがCmBusMstIntfクラスを継承するものとして宣言する。これにより、このクラスCUserBusMstIntfはCmBusMstIntfクラスと同じ関数を持つことになる。次に、これらの関数をオーバーライドして、所望の動作を実現する新たなクラスCUserBusMstIntfを作成する。

【０１８９】

新たな回路クラスを作成する場合の別の例を示す。ユーザが新たなＣＰＵクラスをCUserCpuという名前で作成する場合を考える。まず、新たなクラスCUserCpuがCmCpuクラスを継承するものとして宣言する。これにより、このクラスCUserCpuはCmCpuクラスと同じ関数を持つことになる。次に、これらの関数をオーバーライドして、所望の動作を実現する新たなクラスCUserCpuを作成する。

【０１９０】

[本実施の形態の主な効果]

【０１９１】

効果１：回路モジュールの差し替えが容易である。すなわち、システムを構成する回路モジュールの一部分を差し替えた場合でも、他の回路モジュールの記述を変更する必要がない。

【０１９２】

効果２：回路モジュール間の接続の正しさが自動で検査される。すなわち、回路モジュールを誤って接続すると、コンパイラでコンパイルする際にエラーとなる。

【０１９３】

効果３：情報表示の仕組みを、同じ性質を持つ回路モジュールの間で共通に使用することができる。すなわち、情報表示の仕組みを回路毎に別途に用意する必要はない。

【０１９４】

効果４：シミュレーション制御の仕組みを、シミュレーション対象のシステム間で共通に使用することができる。すなわち、 シミュレーション対象のシステムを変更しても、シミュレーション制御の仕組みを変更する必要がない。

【０１９５】

効果５：回路のインターフェース部の記述が十分であるかどうかを自動で検査できる。すなわち、不足している場合、コンパイラでコンパイルする際にエラーとなる。

【０１９６】

[上記効果が得られる理由]

【０１９７】

○効果１の理由

【０１９８】

効果１の理由の理由について、図３１～図３３を参照して説明する。図３１は、あるクラスClassXの記述例（ａ）と、クラスClassXとクラスCmBusMstIntfとの関係の模式図（ｂ）を示している。図３１（ａ）に示すように、クラスClassXに、回路基底クラスCmBusMstIntfのオブジェクトと接続するための関数ConnectBusMstIntfを定義するとともに（図３１（ａ）の①）、回路基底クラスCmBusMstIntfのオブジェクトBusMstIntfのAppendReadData関数およびStartRead関数を実行する（図３１（ａ）の②，③）。これによって、図３１（ｂ）に示すように、クラスClassXのオブジェクトは、回路基底クラスCmBusMstIntfを継承したクラスのオ

ブジェクトのAppendReadData関数およびStartRead関数を実行することが可能となる。

【０１９９】

例えば、クラスClassXのオブジェクトObjectXと、回路基底クラスCmBusMstIntfの派生クラスであるクラスCmPciBusMstIntfのオブジェクトPciBusMstIntfとを有するシステムを記述する場合、図３２（ａ）に示すように、オブジェクトObjectXをクラスClassXのオブジェクトとして定義するとともに（図３２（ａ）の①）、オブジェクトPciBusMstIntfをクラスCmPciBusMstIntfのオブジェクトとして定義し（図３２（ａ）の②）、さらに、オブジェクトObjectXをConnectBusMstIntf関数によってオブジェクトPciBusMstIntfに接続する（図３２（ａ）の③）。これによって、図３２（ｂ）に示すように、オブジェクトObjectXは、オブジェクトPciBusMstIntfのAppendReadData関数およびStartRead関数を実行することが可能となる。

【０２００】

また、図３３（ａ）および（ｂ）に示すように、オブジェクトObjectXに接続するバスマスタインターフェースオブジェクトを、オブジェクトPciBusMstIntfから、共通の回路基底クラスCmBusMstIntfの派生クラスであるクラスCUserBusMstIntfのオブジェクトUserBusMstIntfに変更する場合は、システム記述（図３３（ａ））において、オブジェクトUserBusMstIntfをクラスCUserBusMstIntfとして宣言するとともに、オブジェクトObjectXのConnectBusMstIntf関数を実行する際の引数を、オブジェクトPciBusMstIntfのアドレスからオブジェクトUserBusMstIntfのアドレスに変更するだけでよい。この場合、図３１（ａ）に示すオブジェクトClassXの記述は、変更する必要はない。

【０２０１】

以上のように、回路基底クラスが共通である場合、例えば、すべてのバスマスタインターフェースに対して回路基底クラスCmBusMstIntfが基本クラスとなるように共通に定義されている場合、任意のクラスClassXでは、通信相手のバスマスタインターフェースがCmBusMstIntfクラスであると宣言しておけば、通信相手のバスマスタインターフェースが変更されたとしても、クラスClassXの記述を変更

すること無しに、CmBusMstIntfクラスで定義されている手順によって、クラスClassXのオブジェクトと変更後のバスマスタインターフェースとの通信ができることになる。すなわち、ある回路モジュールXに新たなバスマスタインターフェースクラスCUserBusMstIntfを接続する場合でも、回路モジュールXの記述を変更する必要はなくなる。

【０２０２】

一方、仮に回路基底クラスが存在しない場合には、ある回路クラスXからクラスCUserBusMstIntfへ指示を送る場合には、クラスXから指示を送る先がCUserBusMstIntfクラスのオブジェクトであることを、クラスXの記述中に書かなくてはならない。従って、クラスXの接続先を、従来のバスマスタインターフェースから、CUserBusMstIntfに変更する場合、クラスXの記述を書き換える必要がある。

【０２０３】

次に、図３４～図３６を参照して、上記とは別の例を挙げて説明する。ある回路クラスClassYを考える。クラスClassYは別の回路の端子から値を読み込んで、その値に基づいて動作を行うこととする。この場合、回路基底クラスの考え方によれば、クラスClassYの記述（図３４（ａ））では、値を読み込む先の回路オブジェクトがCmComponentクラスであると記述し（図３４（ａ）の①）、このCmComponentクラスのオブジェクトのGetPinValue関数を実行することによって端子から値を読み込むことになる（図３４（ａ）の②）。ここで、回路基底クラスの間にも継承関係が存在していることに注意する。例えば、図３５（ａ）および（ｂ）に示すクラスCUserCpu（図３５（ａ）の①）がクラスCmCpuを継承しているとすると、クラスCUserCpuはクラスCmComponentも継承していることになる。よって、クラスCmComponentの性質を持つ。したがって、クラスCmComponentで定義されているメンバ関数を使用することで、クラスClassYの記述を変更すること無く、CUserCpuオブジェクトをClassYオブジェクトに接続することが可能となる。

【０２０４】

このように、回路基底クラス間に継承関係が存在していることにより、記述を変更することなく回路モジュールの差し替えができる。

【０２０５】

○効果２の理由

【０２０６】

例えば図３６（ａ）および（ｂ）に示すように、バスマスタインターフェースの回路基底クラスCmBusMstIntfの派生クラスであるクラスCUserBusMstIntfを、誤って関数ConnectBusSlvIntfでバススレーブクラスCmBusSlaveのオブジェクトBusSlaveに接続しようとした場合を考える。この場合、ConnectBusSlvIntf関数はCmBusSlvIntfクラスを基本クラスに持つクラスしか接続することができない。CmBusSlvIntfクラスを基本クラスに持たないクラスを接続したコードをコンパイルしようとしても、Ｃ＋＋の仕様により、コンパイル時の型チェックで、Ｃ＋＋コンパイラがエラーを出力する。よって、クラスCmBusSlvIntfを継承していないクラスCUserBusMstIntfを誤ってクラスCmBusSlaveに接続しようとしたことが正しく検出される（図３６（ａ）の①）。このように、誤って接続しようとしたときに必ずエラーが発生するので、ユーザが正しい接続をする手助けとなる。

【０２０７】

○効果３の理由

【０２０８】

ＣＰＵをモデルとするオブジェクトCpuの情報表示部は、GetPc関数やGetReg関数などのオブジェクトCpuの内部状態を取り出す関数によって得られる情報を、加工して、ユーザに内部情報として表示できるようにしている。この情報表示部を、回路基底クラスCmCpu内で定義するとともに、オブジェクトCpuがCmCpuクラスであると記述しておく。すると、CmCpuクラスを継承して作られた回路Cpuは、すべてこの情報処理部をそのまま使用することができることになる。例えば新たなユーザ作成クラスCUserCpuがクラスCmCpuを継承して作られているとすると、クラスCmCpuで定義されている既存のＣＰＵの情報処理部をそのまま用いることができるようになる。つまり、新たなＣＰＵクラスCUserCpu用に新たに情報表示部を用意する必要はない。

【０２０９】

○効果４の理由

【０２１０】

一般に、システムはクロックを持つ順序回路（クロック同期回路）となるので、クラスCmSyncModuleの性質を持つ。そこで、シミュレーション制御部は、システムがクラスCmSyncModuleであるとして、そのReset関数、OneStep関数などを実行することにより、シミュレーションを制御している。これにより、システムがクラスCmSyncModuleを継承している限り、どのようなシステムに対しても、１つのシミュレーション制御の仕組みを共通に適用することができる。

【０２１１】

○効果５の理由

【０２１２】

例えばバススレーブの場合、ReadBusSlave関数やWriteBusSlave関数は、バスとの値の受け渡しに用いられる関数であり、処理の内容が記述されている必要がある。ここで、バススレーブの回路基底クラスCmBusSlaveでは、ReadBusSlave関数やWriteBusSlave関数が純粋仮想関数となっている。よって、システムを構成するバススレーブの回路クラスでReadBusSlave関数やWriteBusSlave関数の記述漏れがあったとしても、クラスCmBusSlaveを継承していれば、Ｃ＋＋コンパイラでコンパイルを行う際に、純粋仮想関数がオーバーライドされていない旨のエラーが出力される。このように、必要な関数の記述漏れを自動でチェックすることができる。

【０２１３】

なお、上述した実施の形態は、本発明の実施形態の一例であって、本発明の実施の形態は上述したものに限定されない。例えば、使用するオブジェクト指向言語はＣ＋＋言語に限定されるものではなく、ＪＡＶＡ言語等、上述したＣ＋＋言語による継承、クラス等のオブジェクト指向技術を実現するための仕組みと同等のものを有するものであれば、それに変更可能である。また、回路基底クラスは、上記のものに限定されず、任意の回路モジュールの性質を有する新たなクラスを追加したり、あるいは上記のクラスを適宜統合したクラスを作成すること等が可能である。

【０２１４】

また、本発明によるシミュレーション方法あるいはシミュレーション装置は、

コンピュータとそれによって実行されるシミュレーションプログラムとによって実現することができ、そのコンピュータによって実行されるプログラムは、コンピュータ読み取り可能な記録媒体に記録して頒布することが可能である。

【０２１５】

【発明の効果】

本発明によれば、オブジェクト指向言語を用い、オブジェクト指向言語の継承、クラス分け等の特徴を利用して複数モジュールによって構成されるシステムを記述することで、複数モジュールからなるシステムをソフトウェアによってシミュレーションする際に、システムおよび各回路モジュールの記述ならびに各モジュール間の接続に関する記述や、システム、回路モジュールの拡張、変更等を、従来に比べ、より容易にかつ確実に行うことができるようになる。

【図面の簡単な説明】

【図１】　本発明の実施の形態で使用するＣ＋＋言語について説明するための説明図

【図２】　本発明によるシミュレーション方法（装置）の基本構成（本発明の前提となる構成）を示すブロック図

【図３】　図２に示すシステム記述１０４を説明するための説明図であって、Ｃ＋＋言語による記述例（ａ）と、図３（ａ）の記述例に対応する模式図（ｂ）を示している。

【図４】　本発明によるシミュレーション方法（装置）の実施の形態において用いられる回路基底クラスの一覧を示す図

【図５】　本発明の実施の形態における回路基底クラスの階層関係図

【図６】　図４の回路基底クラスを基本クラスとして作成される実回路クラスの一例を示す図

【図７】　本発明によるシミュレーション方法（装置）の実施の形態を示すブロック図

【図８】　本発明の実施の形態におけるシステム記述の例を説明する際に使用する回路システムの例を示すブロック図

【図９】　本実施の形態における図８に示す回路システムのシステム記述の

一例を示す図であって、図９～図１３で一続きのシステム記述を示している。

【図１０】　図９に続くシステム記述例を示す図

【図１１】　図１０に続くシステム記述例を示す図

【図１２】　図１１に続くシステム記述例を示す図

【図１３】　図１２に続くシステム記述例を示す図

【図１４】　図４に示すクラスCmComponentの宣言文の記述例を示す図

【図１５】　図４に示すクラスCmSyncModuleの宣言文の記述例を示す図

【図１６】　図４に示すクラスCmBusMasterの宣言文の記述例を示す図

【図１７】　図４に示すクラスCmBusSlaveの宣言文の記述例を示す図

【図１８】　図４に示すクラスCmBusMstIntfの宣言文の記述例を示す図であって、図１８～図１９で一続きの記述例を示している。

【図１９】　図１８に続く記述例を示す図

【図２０】　図４に示すクラスCmBusSlvIntfの宣言文の記述例を示す図であって、図２０～図２１で一続きの記述例を示している。

【図２１】　図２０に続く記述例を示す図

【図２２】　図４に示すクラスCmBusSyetemの宣言文の記述例を示す図

【図２３】　図４に示すクラスCmHierの宣言文の記述例を示す図

【図２４】　本実施の形態におけるバスを介したデータ転送のシミュレーション例を説明する際に用いるシステムの構成を示すブロック図

【図２５】　図２４のシステムの構成において転送するデータ例を示す図

【図２６】　図２４のシステムを用いて、本実施の形態におけるバスを介したデータ転送のシミュレーション例を説明する際の動作タイミングを示すタイミング図

【図２７】　図２６の時間１における動作を時系列で説明する説明図

【図２８】　図２６の時間２における動作を時系列で説明する説明図

【図２９】　図２６の時間３における動作を時系列で説明する説明図

【図３０】　図２６の時間４における動作を時系列で説明する説明図

【図３１】　本実施の形態における新たなクラスの作成方法を説明する説明図であって、クラスの宣言文の記述例（ａ）と、図３１（ａ）の記述例に対応す

る模式図を示している。

【図３２】　図３１に示すクラスClassXを用いたシステム記述の一例を示す図であって、システムの記述例（ａ）と、図３２（ａ）の記述例に対応する模式図を示している。

【図３３】　図３１に示すクラスClassXを用いたシステム記述の他の例を示す図であって、システムの記述例（ａ）と、図３３（ａ）の記述例に対応する模式図を示している。

【図３４】　本実施の形態におけるクラスの他の作成例を説明する説明図であって、クラスの宣言文の記述例（ａ）と、図３４（ａ）の記述例に対応する模式図を示している。

【図３５】　図３４に示すクラスClassYを用いたシステム記述の他の例を示す図であって、システムの記述例（ａ）と、図３５（ａ）の記述例に対応する模式図を示している。

【図３６】　本実施の形態におけるシステムの記述例を説明する説明図であって、システムの記述例（ａ）と、図３６（ａ）の記述例に対応する模式図を示している。

【図３７】　従来のＨＤＬによる回路システムの記述例を説明する説明図であって、回路システムの記述例（ａ）と、図３７（ａ）の記述例に対応する模式図を示している。

【符号の説明】

１０１　ユーザ作成ＣＰＵクラス

１０２　ユーザ作成バスマスタI/Fクラス

１０３　ユーザ作成バスクラス

１０４　システム記述

１０５　ＣＰＵ情報表示部

１０６　バス情報表示部

１０７　シミュレーション制御部

１０８　シミュレータソース

１０９　Ｃ＋＋コンパイラ

１１０　シミュレータ

３００　クラスライブラリ

３０１　ＣＰＵ回路基底クラス

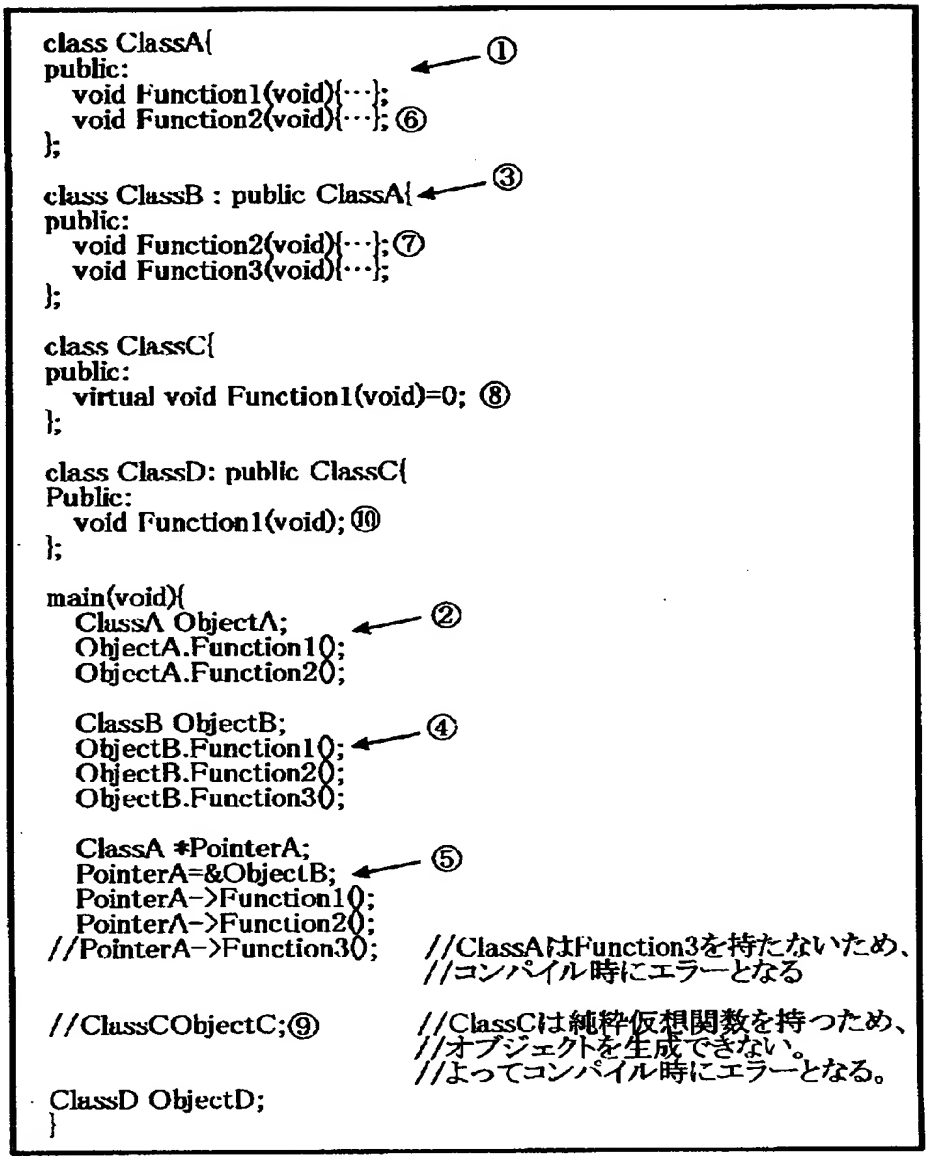
３０２　バスマスタI/F回路基底クラス

３０３　バス回路基底クラス

３０４　同期回路回路基底クラス

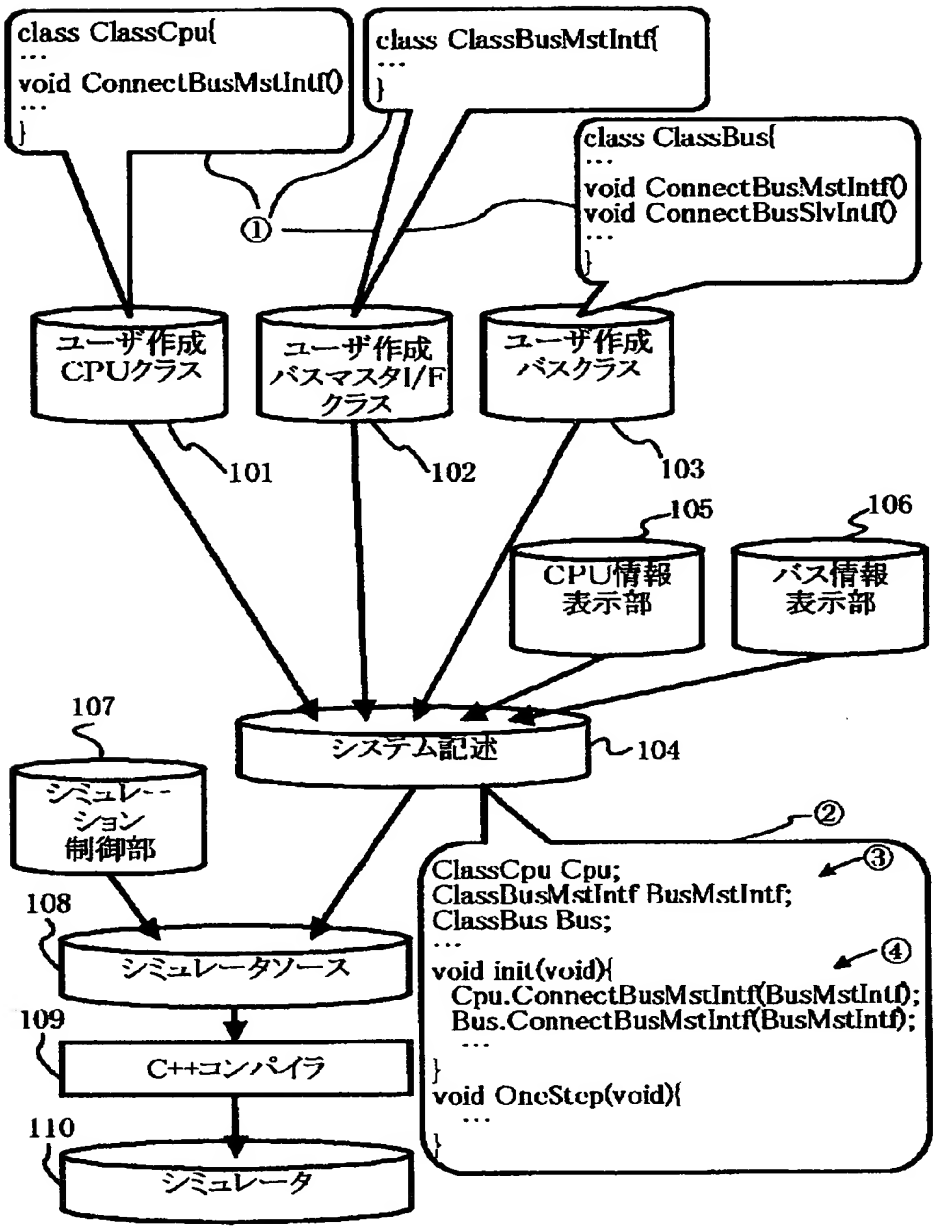
【書類名】　　図面

【図１】



```
class ClassA{                                    ←①
public:
  void Function1(void){…};
  void Function2(void){…}; ⑥
};

class ClassB : public ClassA{                    ←③
public:
  void Function2(void){…};⑦
  void Function3(void){…};
};

class ClassC{
public:
  virtual void Function1(void)=0; ⑧
};

class ClassD: public ClassC{
Public:
  void Function1(void); ⑩
};

main(void){
  ClassA ObjectA;                                ←②
  ObjectA.Function1();
  ObjectA.Function2();

  ClassB ObjectB;                                ←④
  ObjectB.Function1();
  ObjectB.Function2();
  ObjectB.Function3();

  ClassA *PointerA;                              ←⑤
  PointerA=&ObjectB;
  PointerA->Function1();
  PointerA->Function2();
//PointerA->Function3();     //ClassAはFunction3を持たないため、
                             //コンパイル時にエラーとなる

//ClassCObjectC;⑨           //ClassCは純粋仮想関数を持つため、
                             //オブジェクトを生成できない。
                             //よってコンパイル時にエラーとなる。
ClassD ObjectD;
}
```

【図２】


class ClassCpu{
…
void ConnectBusMstIntf()
…
}
class ClassBusMstIntf{
…
}
class ClassBus{
…
void ConnectBusMstIntf()
void ConnectBusSlvIntf()
…
}
①
ユーザ作成 CPUクラス
ユーザ作成 バスマスタI/F クラス
ユーザ作成 バスクラス
101
102
103
105
106
CPU情報 表示部
バス情報 表示部
107
システム記述
104
シミュレーション 制御部
②
③
④
ClassCpu Cpu;
ClassBusMstIntf BusMstIntf;
ClassBus Bus;
…
void init(void){
Cpu.ConnectBusMstIntf(BusMstIntf);
Bus.ConnectBusMstIntf(BusMstIntf);
…
}
void OneStep(void){
…
}
108
シミュレータソース
109
C++コンパイラ
110
シミュレータ

【図３】

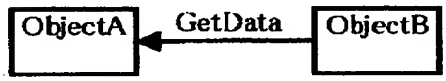
(a)

C++記述の例

```
class ClassA{
   ...
   void GetData(int i);
   ...
};

class ClassB{
   ...
   ClassA *PointerA;
   void ConnectA(ClassA *a)
     {PointerA=a};
   void OneStep(void){
      ...
      ...=PointerA->GetData(i);
      ...
   }
};

main(){
   ClassA ObjectA;
   ClassB ObjectB;
   ObjectB.ConnectA(&ObjectA);
   ...
   ObjectB.OneStep();
   ...
}
```

(b)

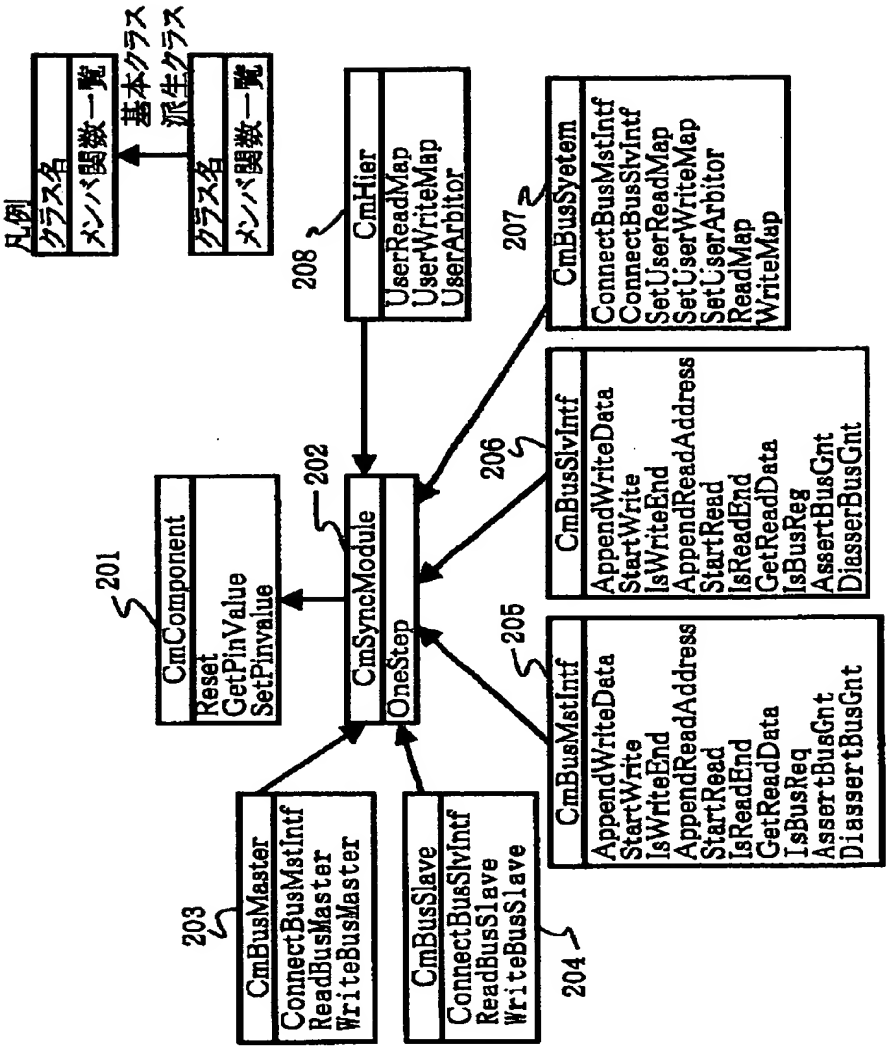
模式図



【図４】

クラス	性質
CmComponent	回路全般
CmSyncModule	クロックに同期して動作する回路
CmBusMaster	バスマスタのメイン部
CmBusSlave	バススレーブのメイン部
CmBusMstIntf	バスマスタインターフェース
CmBusSlvIntf	バススレーブインターフェース
CmBusSystem	バス
CmCpu	CPU
CmMemory	メモリ
CmHier	バスを含む回路の1階層

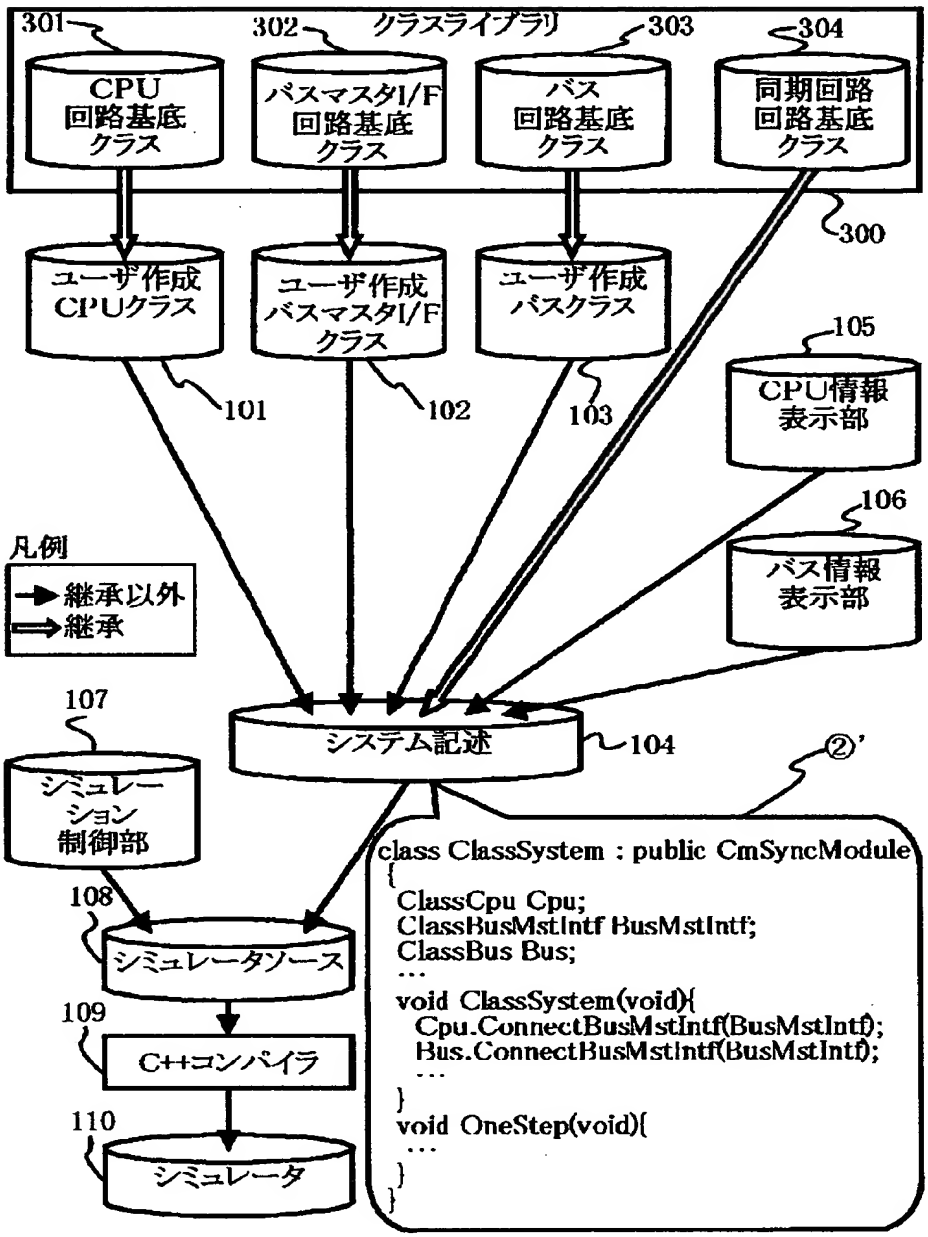
【図５】



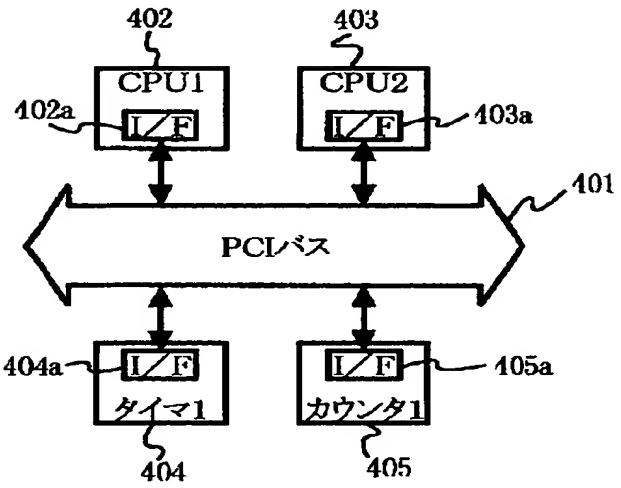
【図６】

実回路クラス	回路基底クラス
CmPciBusMstIntf	CmBusMstIntf
CmPciBusSlvIntf	CmBusSlvIntf
CmPciBusSystem	CmBusSystem
CmV850	CmCpu

【図７】


301
302
クラスライブラリ
303
304
CPU
回路基底
クラス
バスマスタI/F
回路基底
クラス
バス
回路基底
クラス
同期回路
回路基底
クラス
300
ユーザ作成
CPUクラス
ユーザ作成
バスマスタI/F
クラス
ユーザ作成
バスクラス
105
CPU情報
表示部
101
102
103
106
バス情報
表示部
凡例
継承以外
継承
107
システム記述
104
②'
シミュレー
ション
制御部
class ClassSystem : public CmSyncModule
{
ClassCpu Cpu;
ClassBusMstIntf BusMstIntf;
ClassBus Bus;
…
void ClassSystem(void){
Cpu.ConnectBusMstIntf(BusMstIntf);
Bus.ConnectBusMstIntf(BusMstIntf);
…
}
void OneStep(void){
…
}
}
108
シミュレータソース
109
C++コンパイラ
110
シミュレータ

【図８】



【図９】

```
class ABC : public CmHier {    ①

  // システムABCの中にPCIバスを作る。  ②
  CmPciBusSystem PciBusSystem;

  // システムABCの中にCpu1, Cpu2という2つのCPUを作る。  ③
  CmCpu Cpu1, Cpu2;

  // Cpu1,Cpu2はバスマスタなので、システムABCの    ④
  // 中に、Cpu1,Cpu2をバスに繋ぐためのバスマスタ
  // インターフェースを作る。
  CmPciBusMstIntf Cpu1BusMstIntf, Cpu2BusMstIntf;

  // システムABCの中にTimer1という回路を作る。  ⑤
  CmTimer Timer1;

  // Timer1はバススレーブなので、システムABCの    ⑥
  // 中に、Timer1をバスに繋ぐためのバススレーブ
  // インターフェースを作る。
  CmPciBusSlvIntf Timer1BusSlvIntf;

  // システムABCの中にCounter1という回路を作る。  ⑦
  CmCounter Counter1;

  // Counter1はバススレーブなので、システムABCの    ⑧
  // 中に、Counter1をバスに繋ぐためのバススレーブ
  // インターフェースを作る。
  CmPciBusSlvIntf Counter1BusSlvIntf;

  ... // 他のモジュールについての記述を行う(ここでは省略)。  ⑨
```

【図１０】

```
// UserBusReadMapは、バスリードマップを記述する。①
// 戻り値は、RDATAという構造体。
// RDATA.Statusは、0なら読みだし成功、1なら読みだし失敗。
// RDATA.Dataは、読みだし成功のときだけ、読み出した値を戻す。
// ULONGは32bitの信号値。
RDATA UserBusReadMap(ULONG address, int byte_count) {

// アドレス address の値を元に、値を読み出す回路を決めて、その
// 回路のバススレーブインターフェースの読みだし関数を実行する。
   RDATA v;                                                    ②

   // アドレス100から200までは、Timer1から値を読み出すことにする。
   if ((100 <= address) && (address < 200)) {                  ③

      // 読み出すアドレスを指定する。④
      Timer1BusSlvIntf.AppendReadAddress(0, address-100,
                                                   byte_count);

      // 値を読み出す。⑤
      Timer1BusSlvIntf.StartRead(0);

      // 読み出しが成功したかどうかを調べる。⑥
      if (Timer1.IsReadEnd(0)) {

  // 読みだしが成功した。⑦
  v.Status = 0;

  // 読み出した値を取り出す。⑧
  v.Data = Timer1BusSlvIntf.GetReadData(0);
  return v;
      } else {

  // 読み出しが失敗した。⑨
  v.Status = 1;
  return v;
      }
   }
   // アドレス200から300までは、Counter1から値を読み出す。⑩
   else if ((200 <= address) && (address < 300)) {

      // 以下、Timer1のときと同様。⑪

   } else if ... // 他の回路についても同様。
                                                               ⑫
}
```

【図１１】

```
// UserBusWriteMapは、バスライトマップを記述する。①
// 戻り値は、int。
// 0なら書き込み成功、1なら書き込み失敗。
// ULONGは32bitの信号値。
int UserBusWriteMap(ULONG address, ULONG data,
        int byte_count) {

// アドレス address の値を元に、値を書き込む。 ②
// 回路を決めて、その回路のバススレーブ
// インターフェースの書き込み関数を実行する。
                                                    ③
   // アドレス100から200までは、Timer1へ値を書き込むことにする。
   if ((100 <= address) && (address < 200)) {

      // 書き込むアドレスを指定する。 ④
      Timer1BusSlvIntf.AppendWriteData(0, address-100,
                                                  data, byte_count);

      // 値を書き込む。 ⑤
      Timer1BusSlvIntf.StartWrite(0);

      // 書き込みが成功したかどうかを調べる。⑥
      if (Timer1.IsWriteEnd(0)) {

  // 書き込みが成功した。⑦
  return 0;
      } else {

  //書き込みが失敗した。 ⑧
  return 1;
      }
   }
                                                    ⑨
   // アドレス250から350までは、Counter1から値を書き込む(ここ
   //では、読み出しと書き込みでアドレスが異なる)。
   else if ((250 <= address) && (address < 350)) {

      // 以下、Timer1のときと同様。⑩

   } else if ... // 他の回路についても同様。⑪
}
```

【図１２】

```
// UserArbitorでは、アービタを記述する。①
// 戻り値はvoid(戻り値はなし)。
// バス使用許可はバスマスタインターフェースに直に与える。
void UserArbitor(void) {
                                                    ②
    // バス使用要求もバスマスタインターフェースから直に取り出す。
    int Cpu1BusReq = Cpu1BusMstIntf.IsBusReq();
    int Cpu2BusReq = Cpu2BusMstIntf.IsBusReq();

    // ここではCpu1がCpu2よりも優先とする。③
    if (Cpu1BusReq) {

        // Cpu1がバスを要求していたら、Cpu2がバスを要求しているか
        // どうかに関わらず、Cpu1にバスを使用させる。          ④
        Cpu1BusMstIntf.AssertBusGnt();
        Cpu2BusMstIntf.DiassertBusGnt();

    } else if (Cpu2BusReq) {

        // Cpu1がバスを使用していないときだけ、⑤
        // Cpu2はバスを使用することができる。

        Cpu1BusMstIntf.DiassertBusGnt();
        Cpu2BusMstIntf.AssertBusGnt();   ⑥
    } else {
        Cpu1BusMstIntf.DiassertBusGnt();  ⑦
        Cpu2BusMstIntf.DiassertBusGnt();
    }
}
```

【図１３】

```
// ABC(void)は、システムABCのオブジェクトを
// 作る時に呼ばれる関数(コンストラクタ)。      ①
ABC(void) {

  // バスと回路を接続する。②
  Cpu1.ConnectBusMstIntf(&Cpu1BusMstIntf);
  Cpu2.ConnectBusMstIntf(&Cpu2BusMstIntf);
  Timer1.ConnectBusSlvIntf(&Timer1BusSlvIntf);
  Counter1.ConnectBusSlvIntf(&Counter1BusSlvIntf);
  PciBusSystem.ConnectBusMstIntf(&Cpu1BusMstIntf);
  PciBusSystem.ConnectBusMstIntf(&Cpu2BusMstIntf);
  PciBusSystem.ConnectBusSlvIntf(&Timer1BusSlvIntf);
  PciBusSystem.ConnectBusSlvIntf(&Counter1BusSlvIntf);
  ... // 他の回路も接続する(記述を省略)。③

  // アドレスマップを指定する。上で作ったアドレスマップを、 ④
  // PciBusSystemに登録する。このことで、PciBusSystemは上で
  // 作ったアドレスマップを使用できるようになる。
  PciBusSystem.SetUserReadMap(UserReadMap);
  PciBusSystem.SetUserWriteMap(UserWriteMap);
}

// OneStepでは、システムABCが1単位時間にどのような ⑤
// 動作をするかを記述する。
void OneStep(void) {
   // システムABCは、システムのそれぞれの構成要素が勝手に ⑥
   // 動くことによって動作する。システムABCの階層ですることは、
   // 各システムの構成要素を1単位時間動作させることだけ。

   PciBusSystem.OneStep();   ⑦
   // バスは信号線なので、本来はクロックは必要ないが、
   // ここでは、デバグ機能のためにクロックを与えている。

   Cpu1BusMstIntf.OneStep();   ⑧
   Cpu2BusMstIntf.OneStep();
   Timer1BusSlvIntf.OneStep();
   Counter1BusSlvIntf.OneStep();
   Cpu1.OneStep();
   Cpu2.OneStep();
   Timer1.OneStep();
   Counter1.OneStep();
   ... // その他の回路も動作させる。⑨
  }
}
```

【図１４】

```
class CmComponent {
// 用途: クロックを持つ順序回路を記述する際に用いられる。

public:
  virtual void Reset(void) = 0;
    // 用途: 回路の非同期リセットに用いられる。

  virtual RDATA GetPinValue(int no) {
    // 用途: この回路の端子番号noの端子の信号値を戻す。
  }

  virtual int SetPinValue(int no, ULONG data);
    // 用途: この回路の端子番号noの端子に信号値dataを印加する。
  }
};
```

【図１５】

```
class CmSyncModule : public CmComponent {
// 用途: クロックを持つ順序回路を記述する際に用いられる。
// 注意: CmSyncModuleでは、Reset, GetPinValue, SetPinValue
//       について、CmComponentでの定義をそのまま使用する。
public:
  virtual void OneStep(void) = 0;
    // 用途: 回路を1ステップ動作させる。
};
```

【図１６】

```
class CmBusMaster : public CmSyncModule {
// 用途: バスマスタのメイン部を記述するときに使用される。

public:
  virtual void ConnectBusMstIntf(CmBusMstIntf *bus_mst_intf) {
    // 用途: この回路とバスマスタインターフェースを接続する。
  }

  virtual RDATA ReadBusMaster(ULONG address,
     int byte_count) = 0;
    // 用途: この回路からバスへ値を読み出す。

  virtual int WriteBusMaster(ULONG address, ULONG data,
     int byte_count) = 0;
    // 用途: このオブジェクトへバスから来た値を書き込む。
};
```

【図１７】

```
class CmBusSlave : public CmSyncModule {
// 用途: バススレーブのメイン部を記述するときに使用される。

public:
  virtual void ConnectBusSlvIntf(CmBusMstIntf *bus_mst_intf) {
    // 用途: この回路とバススレーブインターフェースを接続する。
  }

  virtual RDATA ReadBusSlave(ULONG address, int byte_count) = 0;
    // 用途: この回路からバスへ値を読み出す。

  virtual int WriteBusSlave(ULONG address, ULONG data,
                                                int byte_count) = 0;
    // 用途: このオブジェクトへバスから来た値を書き込む。
};
```

【図１８】

```
class CmBusMstIntf : public CmSyncModule {
// 用途: バスマスタインターフェースを作成する際に使用される。

public:
  virtual bool IsBusReq(void) {
    // 用途: このバスマスタインターフェースがバスの使用を要
    //     求しているかどうかを戻す。

  }
  virtual void AssertBusGnt(void) {
    // 用途: 外部の回路(通常アービタ)からこのバスマスタインター
    //     フェースに対し、バスの使用を許可すること を通知する。
  }
  virtual void DiassertBusGnt(void) {
    // 用途: 外部の回路(通常アービタ)からこのバスマスタインター
    //     フェースに対し、バスを使用を許可しないことを通知する。
  }
  virtual void AppendWriteData(int mode, ULONG address,
                                        ULONG data, int byte_count) {
    // 用途: modeが0のときは、バスを介して回路モジュールに書
    //       き込むアドレスaddress、データdata, バイト数byte_count
    //       の組を、この回路内部のバッファに登録する。
  }
  virtual void StartWrite(int mode) {
    // 用途: modeが0のときは、バスへの書き込み動作を開始する。
  }
  virtual bool IsWriteEnd(int mode) {
    // 用途: modeが0のときは、バスへの書き込み動作が終了した
    //       かどうかを戻す。
  }
  virtual void AppendReadAddress(int mode,
                                    ULONG address,  int byte_count) {
    // 用途: modeが0のときは、バスを介して回路モジュールから
    //       読み出すアドレスaddress,バイト数byte_countの組を、
    //       この回路内部のバッファに登録する。
  }
```

【図１９】

(class CmBusMstIntf の定義のつづき)

```
  virtual void StartRead(int mode) {
    // 用途: modeが0のときは、バスからの読み出し動作を開始する。
  }
  virtual bool IsReadEnd(int mode) {
    // 用途: modeが0のときは、バスからの読み出し動作が終了し
    //       たかどうかを戻す。
  }
  virtual ULONG GetReadData(int mode){
    // 用途:modeが0のとき、かつバスからの読み出し動作が終了し
    //       たとき、バスから読み出してバッファに格納していた
    //       データから1つを取り出して、バスマスタへ戻す。
  }
  virtual void OneStep(void) {
    // 用途: この回路を1ステップ動作させる。
    // 動作: バスマスタからバスへの読み書き動作中は、バスに
    //       読み書きすべき情報を1つバッファから取り出して、
    //       バスに送る。バッファが空になったら、読み書
    //       き動作を終了する。
    // 補足: CmSyncModuleで定義されたOneStep関数をオーバー
          ライドしている。
  }
};
```

【図２０】

```
class CmBusSlvIntf : public CmSyncModule {
public:
  virtual bool IsBusReq(void) {
    // 用途: このバススレーブインターフェースがバスの使用を
    //       要求しているかどうかを戻す。
  }
  virtual void AssertBusGnt(void) {
    // 用途: 外部の回路(通常アービタ)からこのバススレーブインター
    //       フェースに対し、バスの使用を許可することを通知する。
  }
  virtual void DiassertBusGnt(void) {
    // 用途: 外部の回路(通常アービタ)からこのバススレーブインター
    //       フェースに対し、バスを使用を許可しないことを通知する。
  }
  virtual void AppendWriteData(int mode, ULONG address,
                               ULONG data, int byte_count) {
    // 用途: modeが0のときは、バススレーブに書き込むアドレス
    //       address、データdata, バイト数byte_countの組を、
    //       この回路内部のバッファに登録する。
  }
  virtual void StartWrite(int mode) {
    // 用途: modeが0のときは、バッファに登録された情報を元に
    //       バススレーブへの書き込みを行う。
  }
  virtual bool IsWriteEnd(int mode) {
    // 用途: modeが0のときは、バススレーブへの書き込み動作が
    //       終了したかどうかを戻す。
  }
  virtual void AppendReadAddress(int mode, ULONG address,
                                 int byte_count) {
    // 用途: modeが0のときは、バススレーブから読み出すアドレ
    //       スaddress,バイト数byte_countの組を、この回路内
    //       部のバッファに登録する。
  }
```

【図２１】

class CmBusSlvIntfの定義のつづき

```
  virtual void StartRead(int mode) {
    // 用途: modeが0のときは、バッファに登録された情報を元に
    //       バススレーブからの読み出しを行う。
  }
  virtual bool IsReadEnd(int mode) {
    // 用途: modeが0のときは、バススレーブからの読み出し動作
    //       が終了したかどうかを戻す。
  }
  virtual ULONG GetReadData(int mode){
    // 用途:modeが0のとき、かつバススレーブからの読み出し動作
    //       が終了したとき、バススレーブから読み出してバッファに
    //       格納していたデータから1つを取り出して、バスへ戻す。
  }
  virtual void OneStep(void) {
    // 用途: この回路を1ステップ動作させます。
    // 動作: バススレーブへの読み書きに関しては、OneStepは何
    //       もしないん。
    // 補足: CmBusSlvIntfにはSyncModuleで定義されたOneStep関
    //       数をオーバーライドしている。
  }
};
```

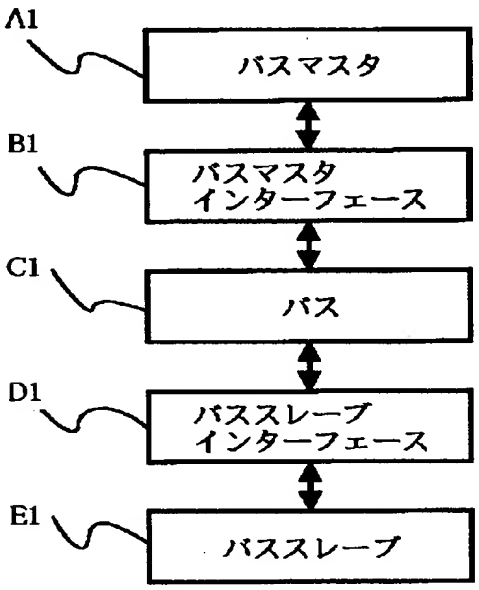
【図２２】

```
class CmBusSystem : public CmSyncModule {
public:
  virtual void ConnectBusMstIntf(CmBusMstIntf *bus_mst_intf) {
    // 用途: バスにバスマスタインターフェースを接続する。
  }
  virtual void ConnectBusSlvIntf(CmBusSlvIntf *bus_slv_intf) {
    // 用途: バスにバススレーブインターフェースを接続する。
  }
  virtual void SetUserReadMap(CmHier *hier) {
    // 用途: バスリードマップを指定します。CmHierクラスのオブジェクト
    //       を渡すと、このCmHierオブジェクトのUserReadMap関数が
    //       このバスのバスリードマップとして登録される。
  }
  virtual void SetUserWriteMap(Cml lier *hier) {
    // 用途: バスライトマップを指定します。CmHierクラスのオブジェクト
    //       を渡すと、このCmHierオブジェクトのUserWriteMap関数が
    //       このバスのバスライトマップとして登録される。
  }
  virtual void SetUserArbitor(Cml lier *hier) {
    // 用途: アービタを指定します。Cml lierクラスのオブジェク
    //       トを渡すと、このCmHierオブジェクトのUserArbitor
    //       関数がこのバスのアービタとして登録される。
  }
  virtual RDATA ReadMap(ULONG address, int byte_count) {
    // 用途: バスを介して回路モジュールから情報を読み出す。
    //       与えられたアドレスaddressを元に読み出しの対象となる回路
    //       モジュールを特定して、その回路モジュールからバイト数
    //       byte_count分のデータを読み出し、読み出したデータを戻す。
  }
  virtual int WriteMap(ULONG address, ULONG data, int byte_count) {
    // 用途: バスを介して回路モジュールへ情報を書き込みます。
    //       与えられたアドレスaddressを元に書き込みの対象と
    //       なる回路モジュールを特定して、その回路モジュー
    //       ルへデータdataをバイト数byte_count分だけ書き込む。
    //       正常に書き込めたかどうかを戻す。
  }
  virtual void OneStep(void) {
    // 用途: バスアービタを1ステップ動作させる。
    //       また、バスの情報表示機能を1ステップ動作させる。
    // 補足: バス本体はただの信号線なので、クロックが来たこ
    //       とに対応して何か動作をする、ということはない。
  }
};
```

【図２３】

```
class CmHier : public CmSyncModule {
  virtual RDATA UserReadMap(ULONG address, int byte_count) {
    // 用途: このシステムに含まれるバスのバスリードマップを
    //       記述する際に使用される。
  }
  virtual int UserWriteMap(ULONG address, ULONG data,
                                                int byte_count) {
    // 用途: このシステムに含まれるバスのバスライトマップを
    //       記述する際に使用される。
  }
  virtual void UserArbitor(void) {
    // 用途: このシステムに含まれるバスのアービタを記述する
    //       際に使用される。
  }
}
```

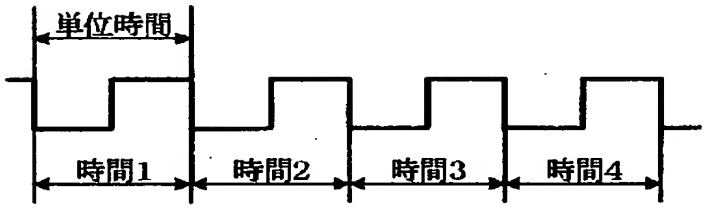
【図２４】



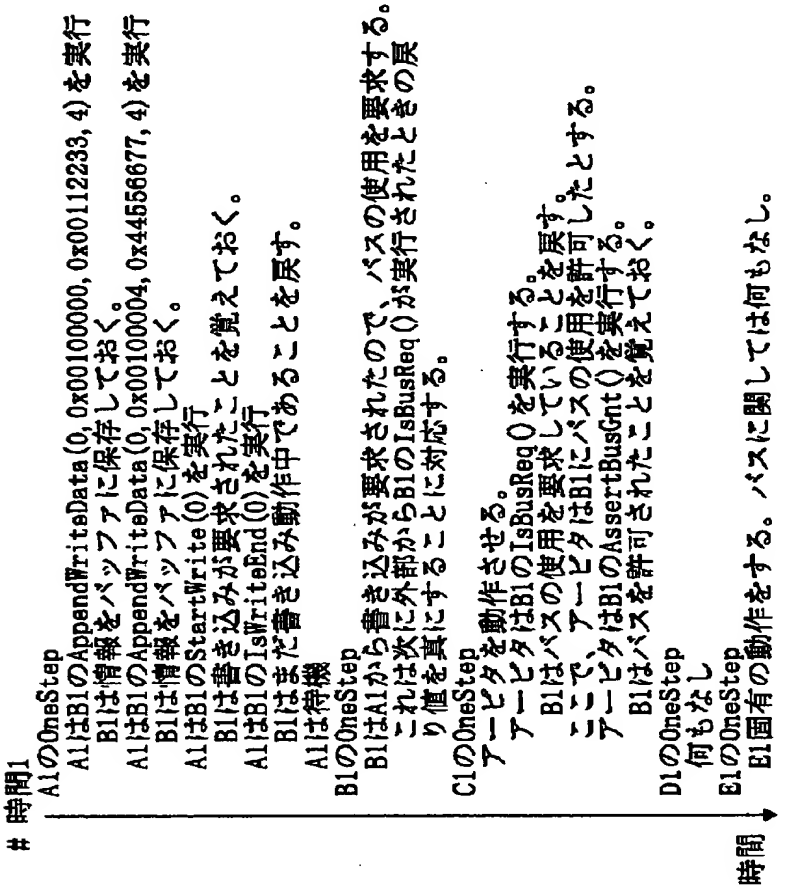
【図２５】

アドレス	データ	バイト数
0x00100000	0x00112233	4
0x00100004	0x44556677	4

【図２６】


単位時間
時間1
時間2
時間3
時間4

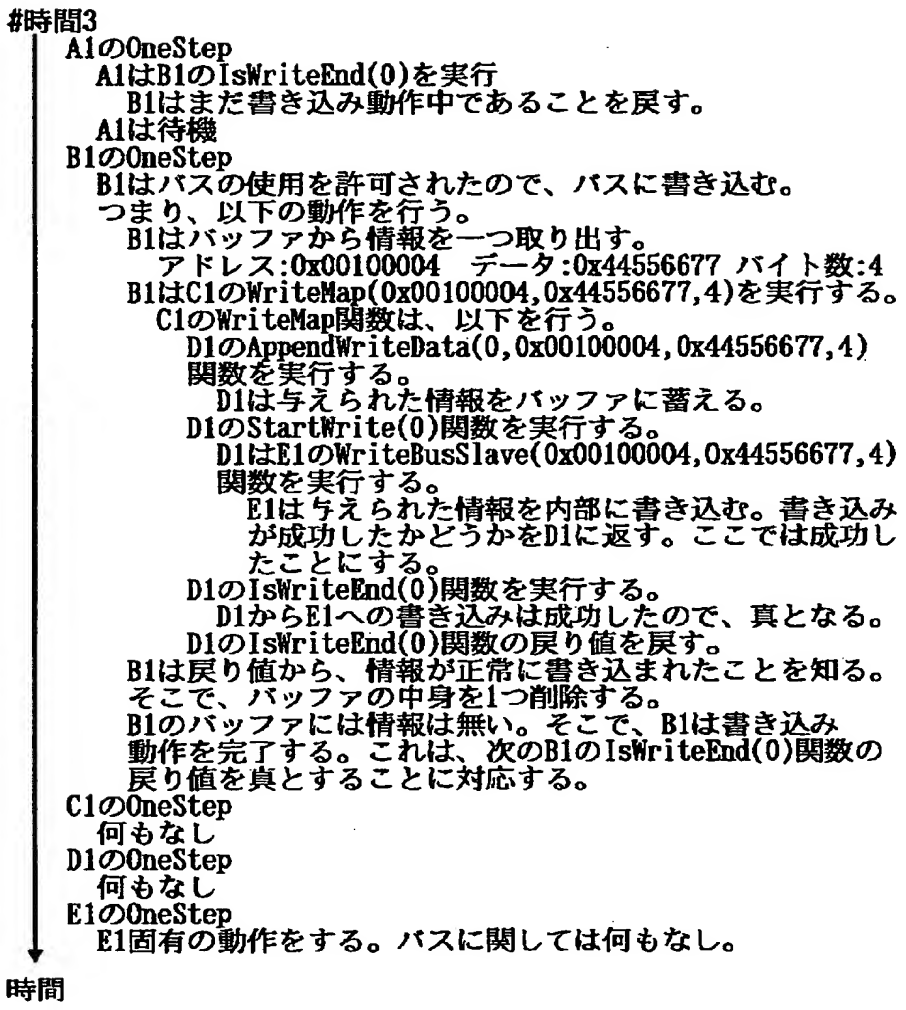
【図２７】


時間1
A1のOneStep
A1はB1のAppendWriteData(0, 0x00100000, 0x00112233, 4)を実行
B1は情報をバッファに保存しておく。
A1はB1のAppendWriteData(0, 0x00100004, 0x44556677, 4)を実行
B1は情報をバッファに保存しておく。
A1はB1のStartWrite(0)を実行
B1は書き込みが要求されたことを覚えておく。
A1はB1のIsWriteEnd(0)を実行
B1はまだ書き込み動作中であることを戻す。
A1は待機
B1のOneStep
B1はA1から書き込みが要求されたので、バスの使用を要求する。
これは次に外部からB1のIsBusReq()が実行されたときの戻り値を真にすることに対応する。
C1のOneStep
アービタを動作させる。
アービタはB1のIsBusReq()を実行する。
B1はバスの使用を要求していることを戻す。
ここで、アービタはB1にバスの使用を許可したとする。
アービタはB1のAssertBusGnt()を実行する。
B1はバスを許可されたことを覚えておく。
D1のOneStep
何もなし
E1のOneStep
E1固有の動作をする。バスに関しては何もなし。
時間

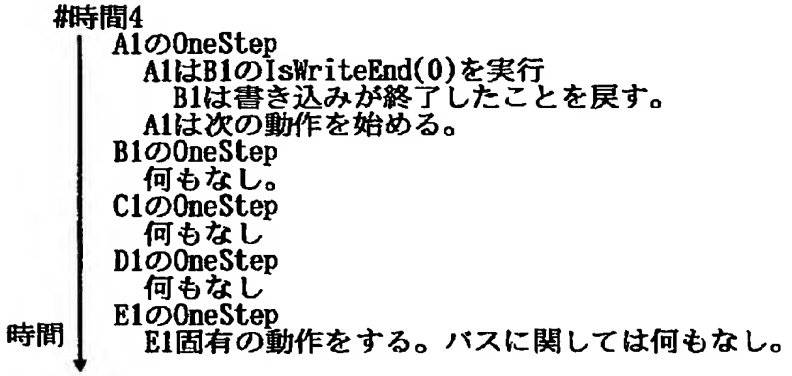
【図２８】

#時間2
A1のOneStep
A1はB1のIsWriteEnd(0)を実行
B1はまだ書き込み動作中であることを戻す。
A1は待機
B1のOneStep
B1はバスの使用を許可されたので、バスに書き込む。
つまり、以下の動作を行う。
B1はバッファから情報を一つ取り出す。
アドレス:0x00100000　データ:0x00112233　バイト数:4
B1はC1のWriteMap(0x00100000,0x00112233,4)を実行する。
C1のWriteMap関数は、以下を行う。
D1のAppendWriteData(0,0x00100000,0x00112233,4)
関数を実行する。
D1は与えられた情報をバッファに蓄える。
D1のStartWrite(0)関数を実行する。
D1はE1のWriteBusSlave(0x00100000,0x00112233,4)
関数を実行する。
E1は与えられた情報を内部に書き込む。書き込みが成功したかどうかをD1に返す。ここでは成功したことにする。
D1のIsWriteEnd(0)関数を実行する。
D1からE1への書き込みは成功したので、真となる。
D1のIsWriteEnd(0)関数の戻り値を戻す。
B1は戻り値から、情報が正常に書き込まれたことを知る。
そこで、バッファの中身を1つ削除する。
B1にはまだバッファに1つ情報があるので、次のB1の
OneStepの実行でも同じことを行う。
C1のOneStep
何もなし
D1のOneStep
何もなし
E1のOneStep
E1固有の動作をする。バスに関しては何もなし。
時間

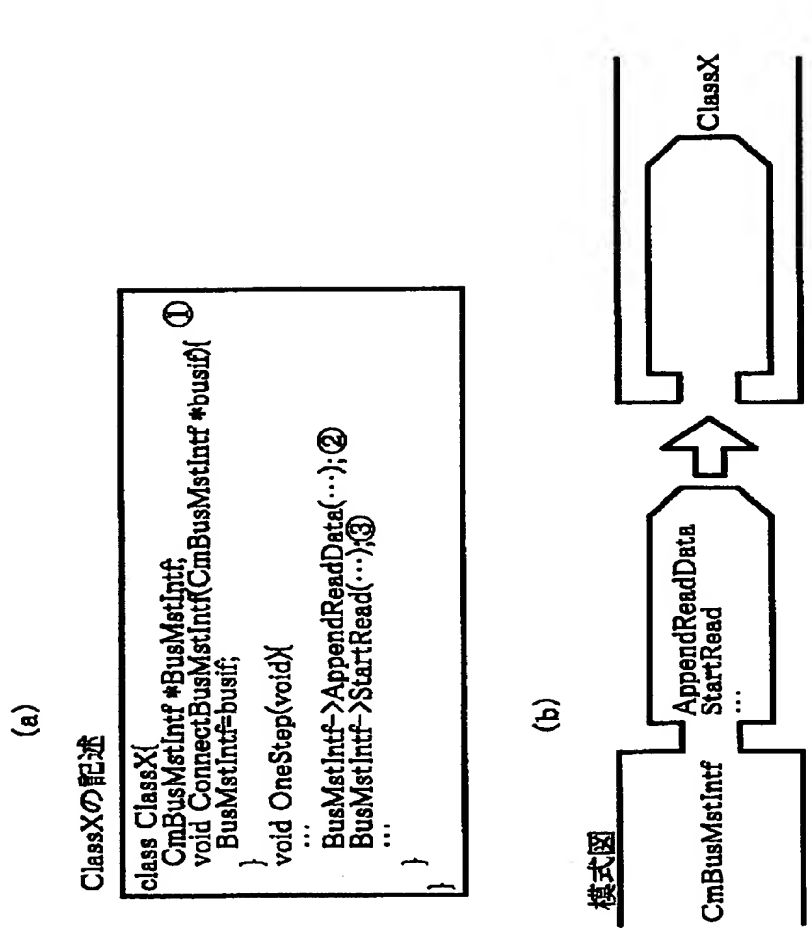
【図２９】


#時間3
A1のOneStep
A1はB1のIsWriteEnd(0)を実行
B1はまだ書き込み動作中であることを戻す。
A1は待機
B1のOneStep
B1はバスの使用を許可されたので、バスに書き込む。
つまり、以下の動作を行う。
B1はバッファから情報を一つ取り出す。
アドレス:0x00100004　データ:0x44556677 バイト数:4
B1はC1のWriteMap(0x00100004,0x44556677,4)を実行する。
C1のWriteMap関数は、以下を行う。
D1のAppendWriteData(0,0x00100004,0x44556677,4)
関数を実行する。
D1は与えられた情報をバッファに蓄える。
D1のStartWrite(0)関数を実行する。
D1はE1のWriteBusSlave(0x00100004,0x44556677,4)
関数を実行する。
E1は与えられた情報を内部に書き込む。書き込み
が成功したかどうかをD1に返す。ここでは成功し
たことにする。
D1のIsWriteEnd(0)関数を実行する。
D1からE1への書き込みは成功したので、真となる。
D1のIsWriteEnd(0)関数の戻り値を戻す。
B1は戻り値から、情報が正常に書き込まれたことを知る。
そこで、バッファの中身を1つ削除する。
B1のバッファには情報は無い。そこで、B1は書き込み
動作を完了する。これは、次のB1のIsWriteEnd(0)関数の
戻り値を真とすることに対応する。
C1のOneStep
何もなし
D1のOneStep
何もなし
E1のOneStep
E1固有の動作をする。バスに関しては何もなし。
時間

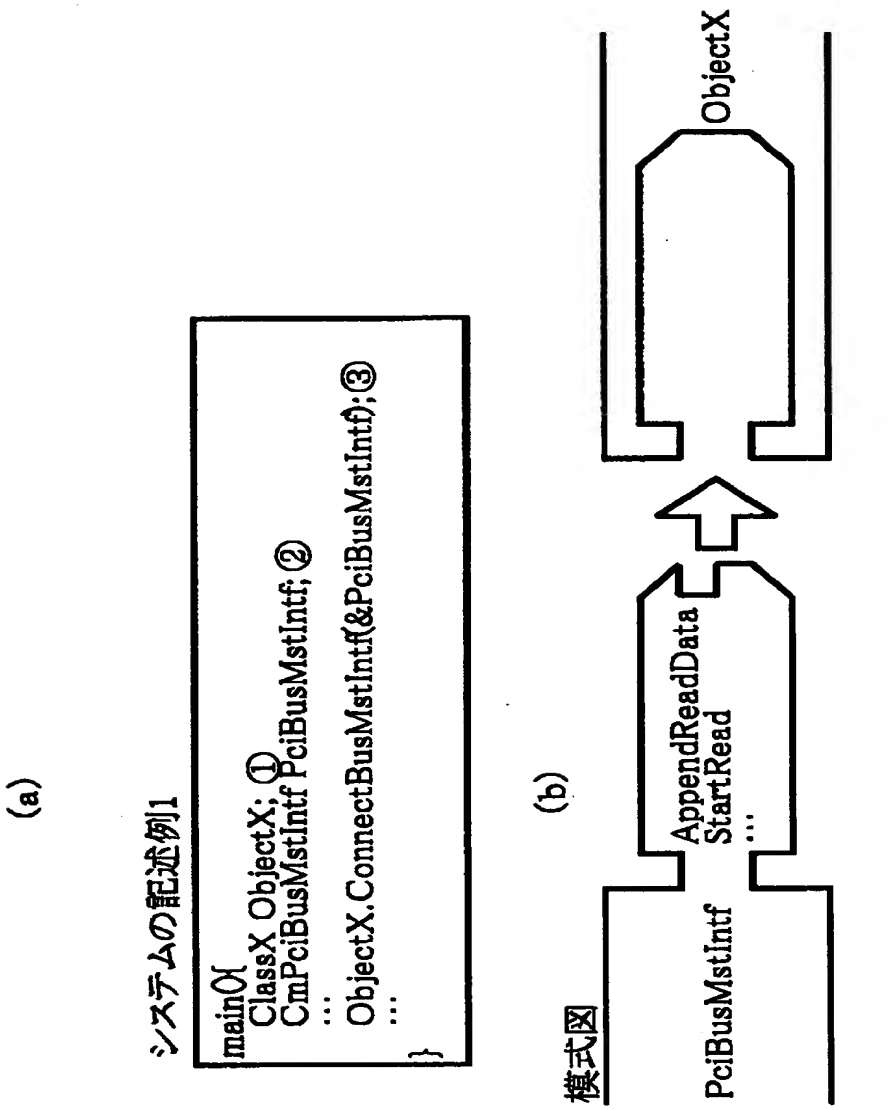
【図３０】


#時間4
A1のOneStep
A1はB1のIsWriteEnd(0)を実行
B1は書き込みが終了したことを戻す。
A1は次の動作を始める。
B1のOneStep
何もなし。
C1のOneStep
何もなし
D1のOneStep
何もなし
E1のOneStep
E1固有の動作をする。バスに関しては何もなし。
時間

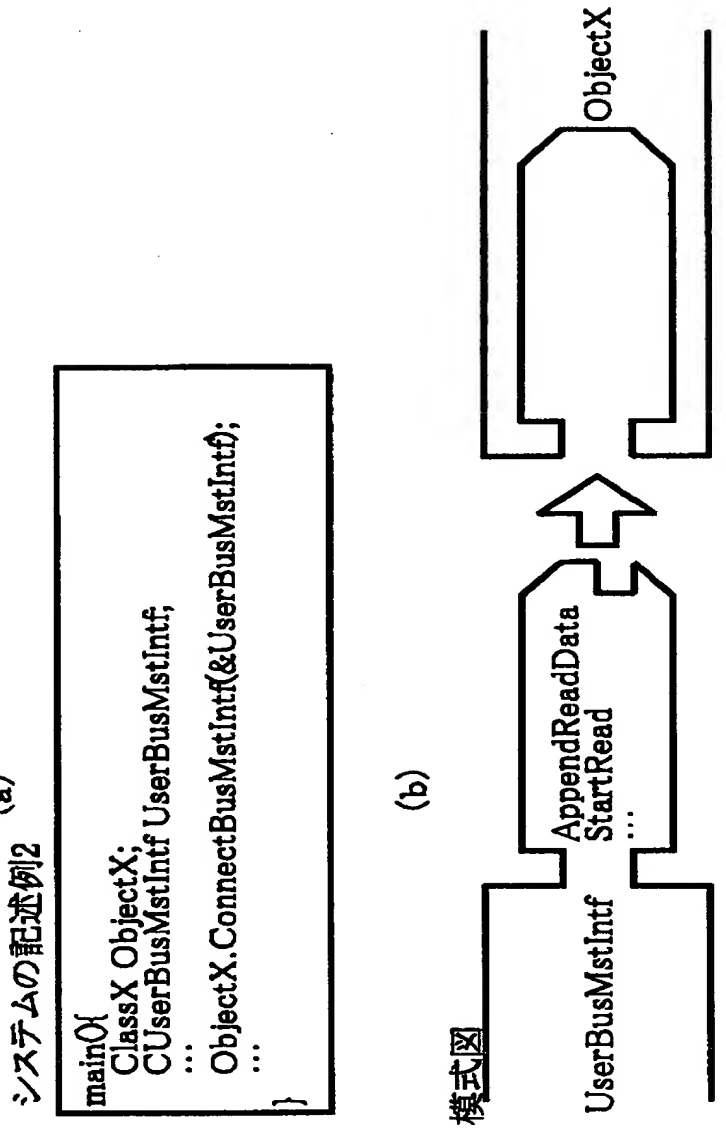
【図３１】


(a)
ClassXの記述
class ClassX{
CmBusMstIntf *BusMstIntf;
void ConnectBusMstIntf(CmBusMstIntf *busif){ ①
BusMstIntf=busif;
}
void OneStep(void){
...
BusMstIntf->AppendReadData(...);②
BusMstIntf->StartRead(...);③
...
}
}
(b)
模式図
CmBusMstIntf
AppendReadData
StartRead
...
ClassX

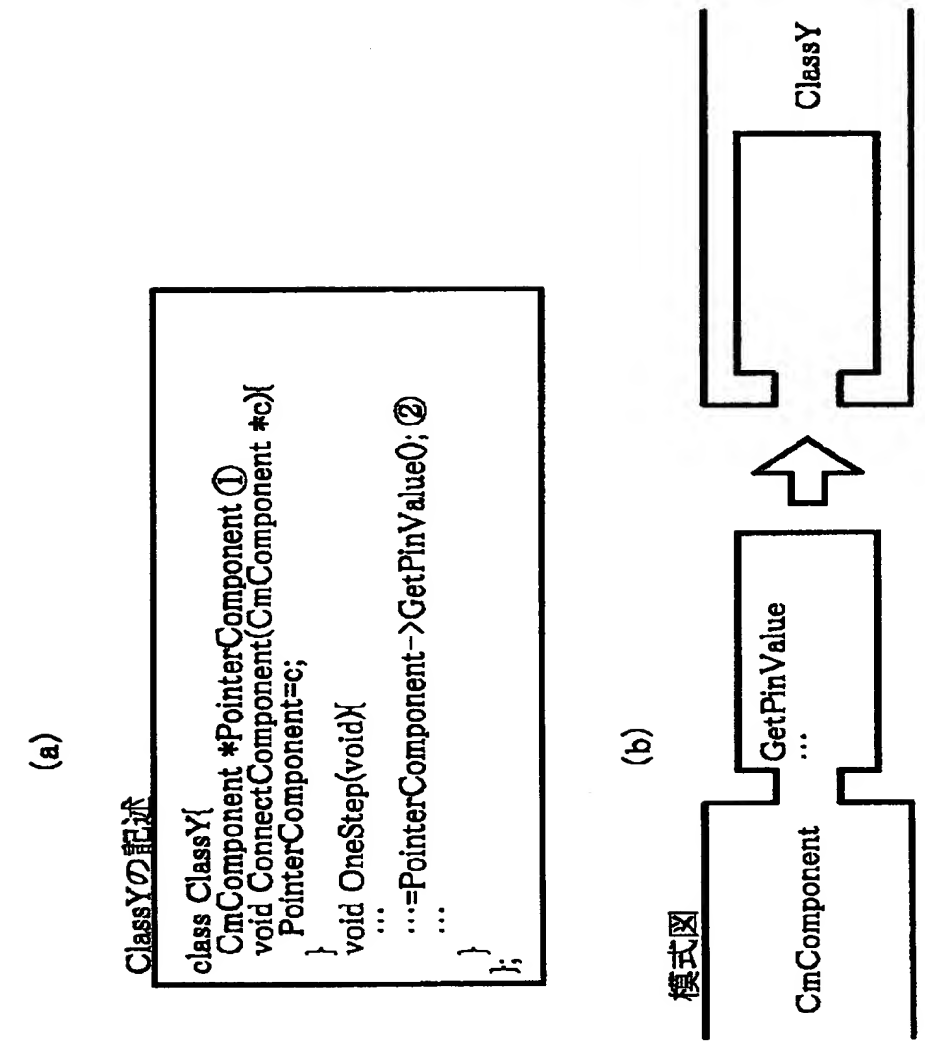
【図３２】


(a)
システムの記述例1
main(){
ClassX ObjectX; ①
CmPciBusMstIntf PciBusMstIntf; ②
...
ObjectX.ConnectBusMstIntf(&PciBusMstIntf);③
...
}
(b)
模式図
PciBusMstIntf
AppendReadData
StartRead
...
ObjectX

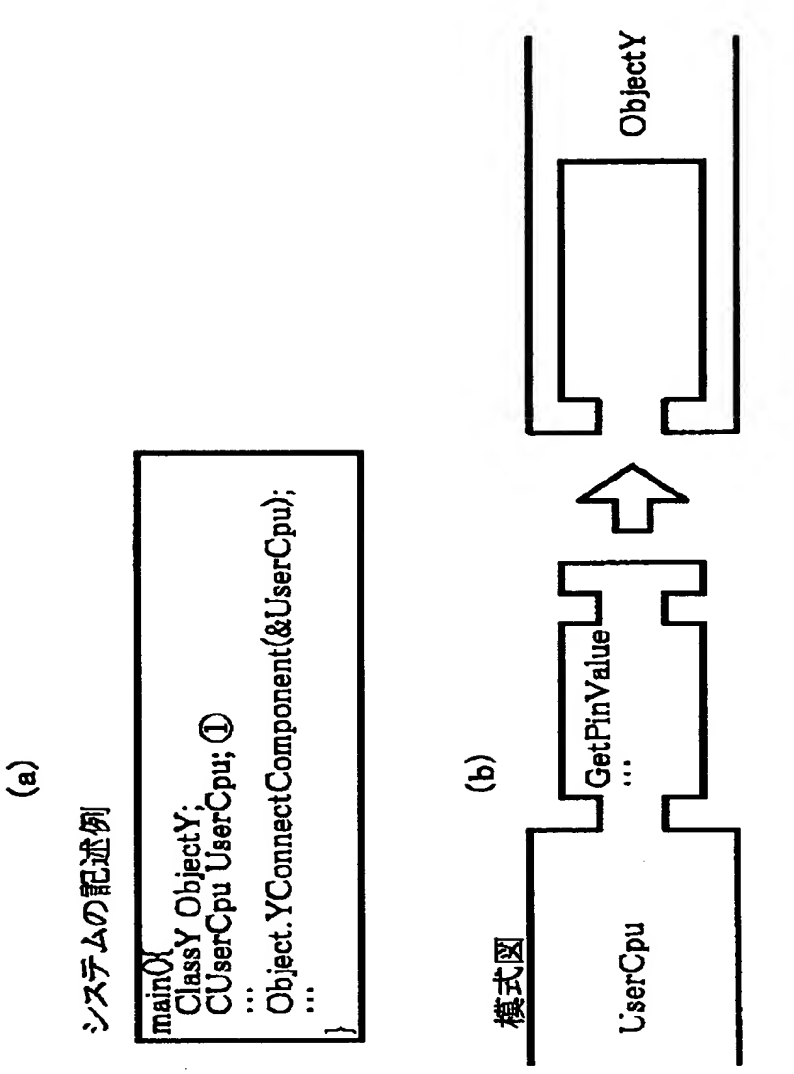
【図３３】


(a)
システムの記述例2
main(){
ClassX ObjectX;
CUserBusMstIntf UserBusMstIntf;
...
ObjectX.ConnectBusMstIntf(&UserBusMstIntf);
...
}
(b)
模式図
UserBusMstIntf
AppendReadData
StartRead
...
ObjectX

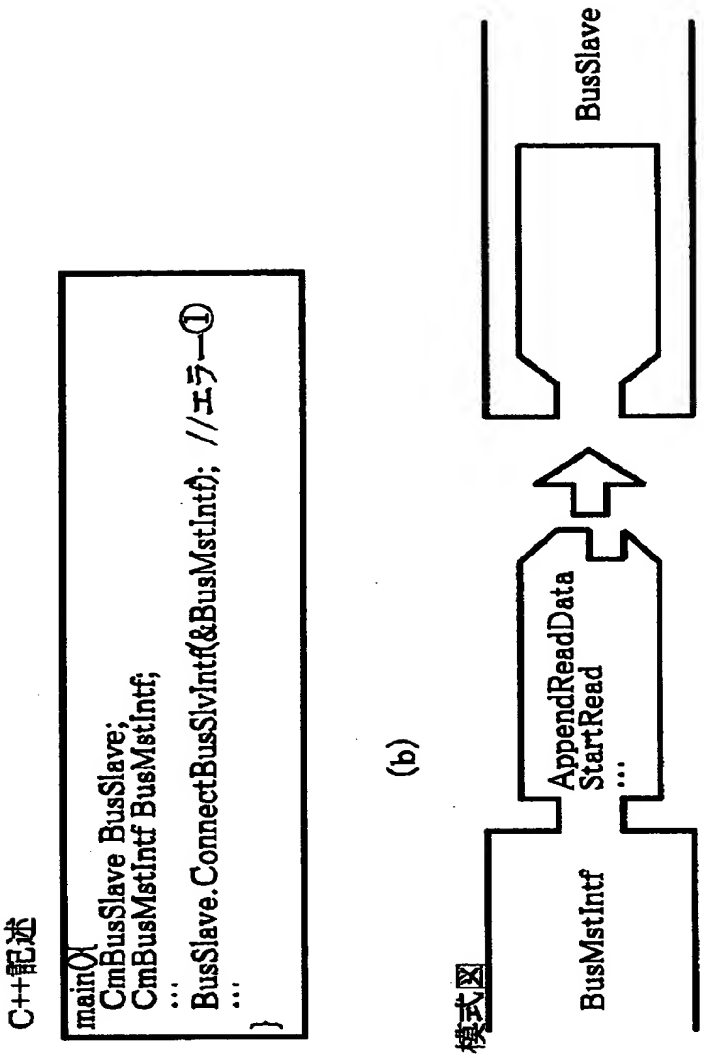
【図３４】


(a)
ClassYの記述
class ClassY{
CmComponent *PointerComponent ①
void ConnectComponent(CmComponent *c){
PointerComponent=c;
}
void OneStep(void){
...
...=PointerComponent->GetPinValue0; ②
...
}
};
(b)
模式図
CmComponent
GetPinValue
...
ClassY

【図３５】


(a)
システムの記述例
main(){
ClassY ObjectY;
CUserCpu UserCpu; ①
:::
Object.YConnectComponent(&UserCpu);
:::
}
(b)
模式図
UserCpu
GetPinValue
:::
ObjectY

【図３６】


(a)
C++記述
main(){
CmBusSlave BusSlave;
CmBusMstIntf BusMstIntf;
...
BusSlave.ConnectBusSlvIntf(&BusMstIntf); //エラー①
...
}
(b)
構式図
BusMstIntf
AppendReadData
StartRead
...
BusSlave

【図３７】

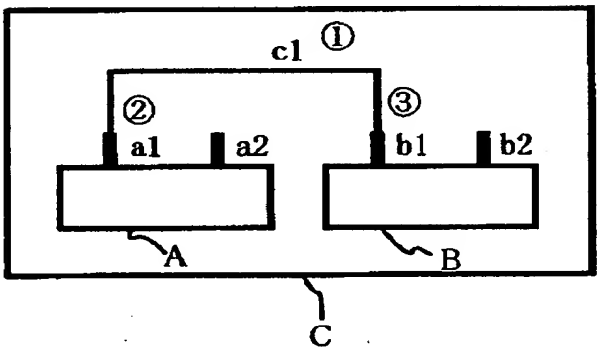
(a)

HDL記述

```
module C;
    wire c1,…
    ModuleA  A(.a1(c1),…);
    ModuleB  B(.b1(c1),…);
endmodule
```

(b)

模式図



【書類名】　　要約書

【要約】

【課題】システムおよび回路モジュールの記述ならびに各モジュール間の接続に関する記述や、システム、回路モジュールの拡張、変更等を、より容易にかつ確実に行うことができるようにする。

【解決手段】複数の回路モジュールからなるシステムをソフトウェアによってシミュレーションするシミュレーション方法において、オブジェクト指向言語を用い、基本となるＣＰＵ、バスマスタＩ／Ｆ等の回路モジュールをクラスとして記述した複数の回路基底クラス301～304をあらかじめクラスライブラリ300として用意しておき、これらの回路基底クラス301～304を継承することで、シミュレーションする回路モジュールをクラス101～103として記述し、クラス101～103として記述した複数の回路モジュールを組み合わせることで、シミュレーションするシステム記述104を作成する。

【選択図】　　図７

認定・付加情報

特許出願の番号	平成１１年　特許願　第３４３９５９号
受付番号	５９９０１１７９０６２
書類名	特許願
担当官	坏　政光　　８８４４
作成日	平成１１年１２月　８日

＜認定情報・付加情報＞

【特許出願人】

【識別番号】　000004237

【住所又は居所】　東京都港区芝五丁目７番１号

【氏名又は名称】　日本電気株式会社

【代理人】　申請人

【識別番号】　100108578

【住所又は居所】　東京都新宿区高田馬場３丁目２３番３号　ＯＲビル　志賀国際特許事務所

【氏名又は名称】　高橋　詔男

【代理人】

【識別番号】　100064908

【住所又は居所】　東京都新宿区高田馬場３丁目２３番３号　ＯＲビル　志賀国際特許事務所

【氏名又は名称】　志賀　正武

【選任した代理人】

【識別番号】　100101465

【住所又は居所】　東京都新宿区高田馬場３丁目２３番３号　ＯＲビル　志賀国際特許事務所

【氏名又は名称】　青山　正和

【選任した代理人】

【識別番号】　100108453

【住所又は居所】　東京都新宿区高田馬場３丁目２３番３号　ＯＲビル　志賀国際特許事務所

【氏名又は名称】　村山　靖彦

次頁無

出　願　人　履　歴　情　報

識別番号　　〔０００００４２３７〕

１．変更年月日　　１９９０年　８月２９日
［変更理由］　新規登録
住　所　東京都港区芝五丁目７番１号
氏　名　日本電気株式会社